Excerpts from "The Linux Users' Guide"

Introductory Course on Scientific Programming

International Master of Science Programme in Scientific Computing

This is an excerpt from the Linux Users' Guide written by Larry Greenfield. It covers basic Unix commands and is intended for the beginning Unix user. The full guide can be downloaded from the home page of The Linux Documentation Project, `http://metalab.unc.edu/LDP/`.

UNIX is a trademark of X/Open
MS-DOS and Microsoft Windows are trademarks of Microsoft Corporation
OS/2 and Operating System/2 are trademarks of IBM
X Window System is a trademark of X Consortium, Inc.
Motif is a trademark of the Open Software Foundation
Linux is not a trademark, and has no connection to UNIX, Unix System Labratories, or to X/Op
Please bring all unacknowledged trademarks to the attention of the author.

Copyright © Larry Greenfield
427 Harrison Avenue
Highland Park, NJ
08904
`leg+@andrew.cmu.edu`

Permission is granted to make and distribute verbatim copes of this manual provided the copyri
notice and this permission notice are preserved on all copies.
Permission is granted to copy and distribute modified versions of this manual under the conditions
verbatim copying, provided also that the sections that reprint "The GNU General Public Licen
"The GNU Library General Public License", and other clearly marked sections held under seper
copyright are reproduced under the conditions given within them, and provided that the en
resulting derived work is distributed under the terms of a permission notice identical to this on
Permission is granted to copy and distribute translations of this manual into another language un
the conditions for modified versions. "The GNU General Public License" and "The GNU Libr
General Public License" may be included in a translation approved by the Free Software Foundat
instead of in the original English.
At your option, you may distribute verbatim and modified versions of this document under
terms of the GNU General Public License, excepting the clearly marked sections held under seper
copyright.

Exceptions to these rules may be granted for various purposes: Write to Larry Greenfield
the above address or email `leg+@andrew.cmu.edu`, and ask. It is requested (but not required) t
you notify the author whenever commercially or large-scale printing this document. Royalties
donations are accepted and will encourage further editions.

These are some of the typographical conventions used in this book.

Bold — Used to mark **new concepts**, **WARNINGS**, and **keywords** in a language.

italics — Used for *emphasis* in text.

slanted — Used to mark **meta-variables** in the text, especially in representations of the command line. For example, "`ls -l` *foo*" where *foo* would "stand for" a filename, such as `/bin/cp`.

`Typewriter` — Used to represent screen interaction.

Also used for code examples, whether it is "C" code, a shell script, or something else, and to display general files, such as configuration files. When necessary for clarity's sake, these examples or figures will be enclosed in thin boxes.

Key — Represents a key to press. You will often see it in this form: "Press return to continue."

◇ — A diamond in the margin, like a black diamond on a ski hill, marks "danger" or "caution." Read paragraphs marked this way carefully.

This X in the margin indicates special instructions for users of the X Window System.

This indicates a paragraph that contains special information that should be read carefully.

Contents

1 The Unix Shell
1.1 Unix Commands
1.1.1 A Typical Unix Command
1.2 Helping Yourself
1.3 Storing Information
1.3.1 Looking at Directories with `ls`
1.3.2 The Current Directory and `cd`
1.3.3 Creating and Removing Directories
1.4 Moving Information
1.4.1 `cp` Like a Monk
1.4.2 Pruning Back with `rm`
1.4.3 A Forklift Can Be Very Handy

2 The X Window System
2.1 Starting and Stopping the X Window System
2.1.1 Starting X
2.1.2 Exiting X
2.2 What is The X Window System?
2.3 What's This on my Screen?
2.3.1 XClock
2.3.2 XTerm
2.4 Window Managers
2.4.1 When New Windows are Created
2.4.2 Focus
2.4.3 Moving Windows
2.4.4 Depth
2.4.5 Iconization
2.4.6 Resizing
2.4.7 Maximization
2.4.8 Menus
2.5 X Attributes
2.5.1 Geometry
2.5.2 Display
2.6 Common Features
2.6.1 Buttons
2.6.2 Menu Bars
2.6.3 Scroll Bars

3 Working with Unix **23**
3.1 Wildcards . . . 23
3.1.1 What *Really* Happens? . . . 23
3.1.2 The Question Mark . . . 24
3.2 Time Saving with `bash` . . . 24
3.2.1 Command-Line Editing . . . 24
3.2.2 Command and File Completion . . . 25
3.3 The Standard Input and The Standard Output . . . 25
3.3.1 Unix Concepts . . . 25
3.3.2 Output Redirection . . . 26
3.3.3 Input Redirection . . . 26
3.3.4 The Pipe . . . 27
3.4 Multitasking . . . 27
3.4.1 Using Job Control . . . 27
3.4.2 The Theory of Job Control . . . 31
3.5 Virtual Consoles: Being in Many Places at Once . . . 32

4 Editing files with Emacs **33**
4.1 What's Emacs? . . . 33
4.2 Getting Started Quickly in X . . . 35
4.3 Editing Many Files at Once . . . 35
4.4 Ending an Editing Session . . . 36
4.5 The Meta Key . . . 37
4.6 Cutting, Pasting, Killing and Yanking . . . 37
4.7 Searching and Replacing . . . 38
4.8 What's Really Going On Here? . . . 38
4.9 Asking Emacs for Help . . . 39
4.10 Specializing Buffers: Modes . . . 40
4.11 Programming Modes . . . 41
4.11.1 C Mode . . . 41
4.11.2 Scheme Mode . . . 41
4.11.3 Mail Mode . . . 42
4.12 Being Even More Efficient . . . 42
4.13 Customizing Emacs . . . 43
4.14 Finding Out More . . . 46

Chapter 1

The Unix Shell

> Making files is easy under the UNIX operating system. Therefore, users tend to create numerous files using large amounts of file space. It has been said that the only standard thing about all UNIX systems is the message-of-the-day telling users to clean up their files.
>
> System V.2 administrator's guide

1.1 Unix Commands

When you first log into a Unix system, you are presented with something that looks like the following:

```
/home/larry#
```

That "something" is called a **prompt**. As its name would suggest, it is prompting you to enter a command. Every Unix command is a sequence of letters, numbers, and characters. There are no spaces, however. Some valid Unix commands are `mail`, `cat`, and `CMU_is_Number-5`. Some characters aren't allowed—we'll go into that later. Unix is also **case-sensitive**. This means that `cat` and `Cat` are different commands.[1]

The prompt is displayed by a special program called the **shell**. Shells accept commands, and run those commands. They can also be programmed in their own language, and programs written in that language are called "shell scripts".

There are two major types of shells in Unix: Bourne shells and C shells. Bourne shells are named after their inventor, Steven Bourne. Steven Bourne wrote the original Unix shell `sh`, and most shells since then end in the letters `sh` to indicate they are extentions on the original idea. There are many implementations of his shell, and all those specific shell programs are called Bourne shells. Another class of shells, C shells (originally implemented by Bill Joy), are also common. Traditionally, Bourne shells have been used for shell scripts and compatibility with the original `sh` while C shells have been used for interactive use. (C shells have had the advantages of having better interactive features but somewhat harder programming features.)

LINUX comes with a Bourne shell called `bash`, written by the Free Software Foundation. `bash` stands for **B**ourne **A**gain **Sh**ell, one of the many bad puns in Unix. It is an "advanced" Bourne shell: it contains the standard programming features found in all Bourne shells with many interactive features commonly found in C shells. `bash` is the default shell to use running LINUX.

When you first login, the prompt is displayed by `bash`, and you are running your first Unix program, the `bash` shell. As long as you are logged in, the `bash` shell will constantly be running.

[1] Case sensitivity is a very personal thing. Some operating systems, such as OS/2 or Windows NT are case preserving, but not case sensitive. In practice, Unix rarely uses the different cases. It is unusual to have a situation where `cat` and `Cat` are different commands.

1.1.1 A Typical Unix Command

The first command to know is `cat`. To use it, type `cat`, and then return:

```
/home/larry# cat
```

If you now have a cursor on a line by itself, you've done the correct thing. There are several variances you could have typed—some would work, some wouldn't.

- If you misspelled `cat`, you would have seen

  ```
  /home/larry# ct
  ct: command not found
  /home/larry#
  ```

  Thus, the shell informs you that it couldn't find a program named "`ct`" and gives you another prompt to work with. Remember, Unix is case sensitive: `CAT` is a misspelling.

- You could have also placed whitespace before the command, like this:[2]

  ```
  /home/larry#␣␣␣␣␣cat
  ```

  This produces the correct result and runs the `cat` program.

- You might also press return on a line by itself. Go right ahead—it does absolutely nothing.

I assume you are now in `cat`. Hopefully, you're wondering what it is doing. No, it is not a game. `cat` is a useful utility that won't seem useful at first. Type anything and hit return. What you should have seen is:

```
/home/larry# cat
Help! I'm stuck in a Linux program!
Help!  I'm stuck in a Linux program!
```

(The *slanted* text indicates what I typed to `cat`.) What `cat` seems to do is echo the text right back at yourself. This is useful at times, but isn't right now. So let's get out of this program and move onto commands that have more obvious benefits.

To end many Unix commands, type Ctrl-d[3]. Ctrl-d is the end-of-file character, or EOF for short. Alternatively, it stands for end-of-text, depending on what book you read. I'll refer to it as an end-of-file. It is a control character that tells Unix programs that you (or another program) is done entering data. When `cat` sees you aren't typing anything else, it terminates.

For a similar idea, try the program `sort`. As its name indicates, it is a sorting program. If you type a couple of lines, then press Ctrl-d, it will output those lines in a sorted order. These types of programs are called **filters**, because they take in text, filter it, and output the text slightly differently. Both `cat` and `sort` are unusual filters. `cat` is unusual because it reads in text and performs *no* changes on it. `sort` is unusual because it reads in lines and doesn't output anything until after it's seen the EOF character. Many filters run on a line-by-line basis: they will read in a line, perform some computations, and output a different line.

[2] The '␣' indicates that the user typed a space.

[3] Hold down the key labeled "Ctrl" and press "d", then let go of both.

1.2 Helping Yourself

The `man` command displays reference pages for the command[4] you specify. For example:

```
/home/larry# man cat

cat(1)                                                    cat(1)

NAME
  cat - Concatenates or displays files

SYNOPSIS
  cat [-benstuvAET] [--number] [--number-nonblank] [--squeeze-blank]
  [--show-nonprinting] [--show-ends] [--show-tabs] [--show-all]
  [--help] [--version] [file...]

DESCRIPTION
  This manual page documents the GNU version of cat ...
```

There's about one full page of information about `cat`. Try running `man` now. Don't expect to understand the manpage given. Manpages usually assume quite a bit of Unix knowledge—knowledge that you might not have yet. When you've read the page, there's probably a little black block at the bottom of your screen similar to "`--more--`" or "`Line 1`". This is the **more-prompt**, and you'll learn to love it.

Instead of just letting the text scroll away, `man` stops at the end of each page, waiting for you to decide what to do now. If you just want to go on, press Space and you'll advance a page. If you want to exit (quit) the manual page you are reading, just press `q`. You'll be back at the shell prompt, and it'll be waiting for you to enter a new command.

There's also a keyword function in `man`. For example, say you're interested in any commands that deal with Postscript, the printer control language from Adobe. Type `man -k ps` or `man -k Postscript`, you'll get a listing of all commands, system calls, and other documented parts of Unix that have the word "ps" (or "Postscript") in their name or short description. This can be very useful when you're looking for a tool to do something, but you don't know it's name—or if it even exists!

1.3 Storing Information

Filters are very useful once you are an experienced user, but they have one small problem. How do you store the information? Surely you aren't expected to type everything in each time you are going to use the program! Of course not. Unix provides **files** and **directories**.

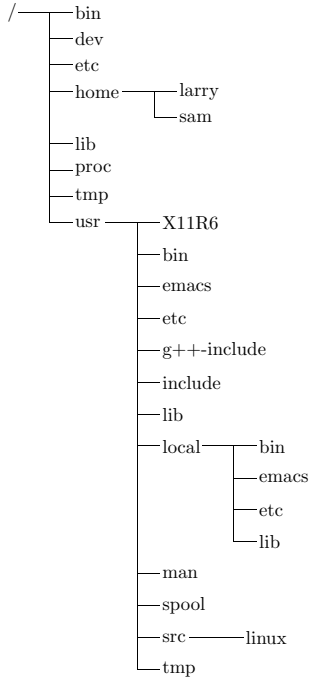
A directory is like a folder: it contains pieces of paper, or files. A large folder can even hold other folders—directories can be inside directories. In Unix, the collection of directories and files is called the file system. Initially, the file system consists of one directory, called the "root" directory. Inside this directory, there are more directories, and inside those directories are files and yet more directories.[5]

Each file and each directory has a name. It has both a short name, which can be the same as another file or directory somewhere else on the system, and a long name which is unique. A short name for a file could be `joe`, while it's "full name" would be `/home/larry/joe`. The full name is usually called the **path**. The path can be decode into a sequence of directories. For example, here is how `/home/larry/joe` is read:

[4] `man` will also display information on a system call, a subroutine, a file format, and more. In the original version of Unix it showed the exact same information the printed documentation would. For now, you're probably only interested in getting help on commands.

[5] There may or may not be a limit to how "deep" the file system can go. (I've never reached it—one can easily have directories 10 levels deep.)

Figure 1.1 A typical (abridged) Unix directory tree.



`/home/larry/joe`
The initial slash indicates the root directory.
This signifies the directory called `home`. It is inside the root directory.
This is the directory `larry`, which is inside `home`.
`joe` is inside `larry`. A path could refer to either a directory or a filename,
so `joe` could be either. All the items *before* the short name must be directories.

An easy way of visualizing this is a tree diagram. To see a diagram of a typical Linux system, look at Figure 1.1. Please note that this diagram isn't complete—a full Linux system has over 8000 files!—and shows only some of the standard directories. Thus, there may be some directories in that diagram that aren't on your system, and your system almost certainly has directories not listed there.

1.3.1 Looking at Directories with `ls`

Now that you know that files and directories exist, there must be some way of manipulating them. Indeed there is. The command `ls` is one of the more important ones. It lists files. If you try `ls` as a command, you'll see:

```
/home/larry# ls
```

```
/home/larry#
```

That's right, you'll see nothing. Unix is intensionally terse: it gives you nothing, not even "no files" if there aren't any files. Thus, the lack of output was `ls`'s way of saying it didn't find any files.

But I just said there could be 8000 or more files lying around: where are they? You've run into the concept of a "current" directory. You can see in your prompt that your current directory is `/home/larry`, where you don't have any files. If you want a list of files of a more active directory, try the root directory:

```
/home/larry# ls /
bin      etc      install  mnt      root     user     var
dev      home     lib      proc     tmp      usr      vmlinux
/home/larry#
```

In the above command, "`ls /`", the directory ("/") is a **parameter**. The first word of the command is the command name, and anything after it is a parameter. Parameters generally modify what the program is acting on—for `ls`, the parameters say what directory you want a list for. Some commands have special parameters called options or switches. To see this try:

```
/home/larry# ls -F /
bin/     etc/     install/  mnt/     root/    user/    var@
dev/     home/    lib/      proc/    tmp/     usr/     vmlinux
/home/larry#
```

The `-F` is an **option**. An option is a special kind of parameter that starts with a dash and modifies how the program runs, but not what the program runs on. For `ls`, `-F` is an option that lets you see which ones are directories, which ones are special files, which are programs, and which are normal files. Anything with a slash is a directory. We'll talk more about `ls`'s features later. It's a surprisingly complex program!

Now, there are two lessons to be learned here. First, you should learn what `ls` does. Try a few other directories that are shown in Figure 1.1, and see what they contain. Naturally, some will be empty, and some will have many, many files in them. I suggest you try `ls` both with and without the `-F` option. For example, `ls /usr/local` looks like:

```
/home/larry# ls /usr/local
archives  bin       emacs     etc       ka9q      lib       tcl
/home/larry#
```

The second lesson is more general. Many Unix commands are like `ls`. They have options, which are generally one character after a dash, and they have parameters. Unlike `ls`, some commands *require* certain parameters and/or options. To show what commands generally look like, we'll use the following form:

`ls` [-aRF] [*directory*]

I'll generally use command templates like that before I introduce any command from now on. The first word is the command (in this case `ls`). Following the command are all the parameters. Optional parameters are contained in brackets ("[" and "]"). Meta-variables are *slanted*—they're words that take the place of actual parameters. (For example, above you see *directory*, which should be replaced by the name of a real directory.)

Options are a special case. They're enclosed by brackets, but you can take any one of them without using all of them. For instance, with just the three options given for `ls` you have eight different ways of running the command: with or without each of the options. (Contrast `ls -R` with `ls -F`.)

1.3.2 The Current Directory and `cd`

`pwd`

Using directories would be cumbersome if you had to type the full path each time you wanted to access a directory. Instead, Unix shells have a feature called the "current" or "present" or "working" directory. Your setup most likely displays your directory in your prompt: `/home/larry`. If it doesn't, try the command `pwd`, for **p**resent **w**orking **d**irectory. (Sometimes the prompt will display the machine name. This is only really useful in a networked environment with lots of different machines.)

```
mousehouse>pwd
/home/larry
mousehouse>
```

`cd` [*directory*]

As you can see, `pwd` tells you your current directory[6]—a very simple command. Most commands act, by default, on the current directory. For instance, `ls` without any parameters displays the contents of the current directory. We can change our current directory using `cd`. For instance, try:

```
/home/larry# cd /home
/home# ls -F
larry/     sam/       shutdown/  steve/     user1/
/home#
```

If you omit the optional parameter *directory*, you're returned to your home, or original, directory. Otherwise, `cd` will change you to the specified directory. For instance:

```
/home# cd
/home/larry# cd /
/# cd home
/home# cd /usr
/usr# cd local/bin
/usr/local/bin#
```

As you can see, `cd` allows you to give either absolute or relative pathnames. An **absolute path** starts with `/` and specifies all the directories before the one you wanted. A **relative path** is in relation to your current directory. In the above example, when I was in `/usr`, I made a relative move to `local/bin`—`local` is a directory under `usr`, and `bin` is a directory under `local`! (`cd home` was also a relative directory change.)

There are two directories used *only* for relative pathnames: "`.`" and "`..`". . The directory "`.`" refers to the current directory and "`..`" is the parent directory. These are "shortcut" directories. They exist in *every* directory, but don't really fit the "folder in a folder" concept. Even the root directory has a parent directory—it's its own parent!

The file `./chapter-1` would be the file called `chapter-1` in the current directory. Occasionally, you need to put the "`./`" for some commands to work, although this is rare. In most cases, `./chapter-1` and `chapter-1` will be identical.

The directory "`..`" is most useful in "backing up":

[6] You'll see all the terms in this book: present working directory, current directory, or working directory. I prefer "current directory", although at times the other forms will be used for stylistic purposes.

```
/usr/local/bin# cd ..
/usr/local# ls -F
archives/  bin/       emacs@     etc/       ka9q/      lib/       tcl@
/usr/local# ls -F ../src
cweb/      linux/     xmris/
/usr/local#
```

In this example, I changed to the parent directory using `cd ..`, and I listed the directory `/usr/src` from `/usr/local` using `../src`. Note that if I was in `/home/larry`, typing `ls -F ../src` wouldn't do me any good!

The directory `~/` is an alias for your home directory:

```
/usr/local# ls -F ~/
/usr/local#
```

You can see at a glance that there isn't anything in your home directory! `~/` will become more useful as we learn more about how to manipulate files.

1.3.3 Creating and Removing Directories

`mkdir` *directory1* [*directory2 ... directoryN*]

Creating your own directories is extremely simple under Unix, and can be a useful organizational tool. To create a new directory, use the command `mkdir`. Of course, `mkdir` stands for **m**a**k**e **dir**ectory.

Let's do a small example to see how this works:

```
/home/larry# ls -F
/home/larry# mkdir report-1993
/home/larry# ls -F
report-1993/
/home/larry# cd report-1993
/home/larry/report-1993#
```

`mkdir` can take more than one parameter, interpreting each parameter as another directory to create. You can specify either the full pathname or a relative pathname; `report-1993` in the above example is a relative pathname.

```
/home/larry/report-1993# mkdir /home/larry/report-1993/chap1 ~/report-1993/chap2
/home/larry/report-1993# ls -F
chap1/  chap2/
/home/larry/report-1993#
```

`rmdir` *directory1* [*directory2 ... directoryN*]

The opposite of `mkdir` is `rmdir` (**r**e**m**ove **dir**ectory). `rmdir` works exactly like `mkdir`.

An example of `rmdir` is:

```
/home/larry/report-1993# rmdir chap1 chap3
rmdir: chap3: No such file or directory
/home/larry/report-1993# ls -F
chap2/
/home/larry/report-1993# cd ..
/home/larry# rmdir report-1993
rmdir: report-1993: Directory not empty
/home/larry#
```


As you can see, `rmdir` will refuse to remove a non-existant directory, as well as a directory that has anything in it. (Remember, `report-1993` has a subdirectory, `chap2`, in it!) There is one more interesting thing to think about `rmdir`: what happens if you try to remove your current directory? Let's find out:

```
/home/larry# cd report-1993
/home/larry/report-1993# ls -F
chap2/
/home/larry/report-1993# rmdir chap2
/home/larry/report-1993# rmdir .
rmdir: .: Operation not permitted
/home/larry/report-1993#
```

Another situation you might want to consider is what happens if you try to remove the parent of your current directory. This turns out not to be a problem since the parent of your current directory isn't empty, so it can't be removed!

1.4 Moving Information

All of these fancy directories are very nice, but they really don't help unless you have some place to store you data. The Unix Gods saw this problem, and they fixed it by giving the users **files**.

We will learn more about creating and editing files in the next few chapters.

The primary commands for manipulating files under Unix are `cp`, `mv`, and `rm`. They stand for **c**o**p**y, **m**o**v**e, and re**m**ove, respectively.

1.4.1 `cp` Like a Monk

`cp` [-i] *source destination*
`cp` [-i] *file1 file2 ... fileN destination-directory*[7]

`cp` is a very useful utility under Unix, and extremely powerful. It enables one person to copy more information in a second than a fourteenth century monk could do in a year.

Be careful with `cp` if you don't have a lot of disk space. No one wants to see a "Disk full" message when working on important files. `cp` can also overwrite existing files without warning—I'll talk more about that danger later.

We'll first talk about the first line in the command template. The first parameter to `cp` is the file to copy—the second is where to copy it. You can copy to either a different filename, or a different directory. Let's try some examples:

```
/home/larry# ls -F /etc/passwd
/etc/passwd
/home/larry# cp /etc/passwd .
/home/larry# ls -F
passwd
/home/larry# cp passwd frog
/home/larry# ls -F
frog  passwd
/home/larry#
```

[7] `cp` has two lines in its template because the meaning of the second parameter can be different depending on the number of parameters.

The first `cp` command I ran took the file `/etc/passwd`, which contains the names of all the users on the Unix system and their (encrypted) passwords, and copied it to my home directory. `cp` doesn't delete the source file, so I didn't do anything that could harm the system. So two copies of `/etc/passwd` exist on my system now, both named `passwd`, but one is in the directory `/etc` and one is in `/home/larry`.

Then I created a *third* copy of `/etc/passwd` when I typed `cp passwd frog`—the three copies are now: `/etc/passwd`, `/home/larry/passwd` and `/home/larry/frog`. The contents of these three files are the same, even if the names aren't.

`cp` can copy files between directories if the first parameter is a file and the second parameter is a directory. In this case, the short name of the file stays the same.

It can copy a file and change it's name if both parameters are file names. Here is one danger of `cp`. If I typed `cp /etc/passwd /etc/group`, `cp` would normally create a new file with the contents identical to `passwd` and name it `group`. However, if `/etc/group` already existed, `cp` would destroy the old file without giving you a chance to save it! (It won't even print out a message reminding you that you're destroying a file by copying over it.)

Let's look at another example of `cp`:

```
/home/larry# ls -F
frog  passwd
/home/larry# mkdir passwd_version
/home/larry# cp frog passwd passwd_version
/home/larry# ls -F
frog         passwd        passwd_version/
/home/larry# ls -F passwd_version
frog  passwd
/home/larry#
```

How did I just use `cp`? Evidentally, `cp` can take *more* than two parameters. (This is the second line in the command template.) What the above command did is copied all the files listed (`frog` and `passwd`) and placed them in the `passwd_version` directory. In fact, `cp` can take any number of parameters, and interprets the first $n-1$ parameters to be files to copy, and the n^{th} parameter as what directory to copy them too.

You cannot rename files when you copy more than one at a time—they always keep their short name. This leads to an interesting question. What if I type `cp frog passwd toad`, where `frog` and `passwd` exist and `toad` isn't a directory? Try it and see.

1.4.2 Pruning Back with `rm`

`rm` [-i] *file1 file2 ... fileN*

Now that we've learned how to create millions of files with `cp` (and believe me, you'll find new ways to create more files soon), it may be useful to learn how to delete them. Actually, it's very simple: the command you're looking for is `rm`, and it works just like you'd expect: any file that's a parameter to `rm` gets deleted.

For example:

```
/home/larry# ls -F
frog         passwd        passwd_version/
/home/larry# rm frog toad passwd
rm: toad: No such file or directory
/home/larry# ls -F
passwd_version/
/home/larry#
```


As you can see, `rm` is extremely unfriendly. Not only does it not ask you for confirmation, but it will also delete things even if the whole command line wasn't correct. This could actually be dangerous. Consider the difference between these two commands:

```
/home/larry# ls -F
toad  frog/
/home/larry# ls -F frog
toad
/home/larry# rm frog/toad
/home/larry#
```

and this

```
/home/larry# rm frog toad
rm: frog is a directory
/home/larry# ls -F
frog/
/home/larry#
```

As you can see, the difference of *one* character made a world of difference in the outcome of the command. It is vital that you check your command lines before hitting return!

1.4.3 A Forklift Can Be Very Handy

`mv` [-i] *old-name new-name*
`mv` [-i] *file1 file2 ... fileN new-directory*

Finally, the other file command you should be aware of is `mv`. `mv` looks a lot like `cp`, except that it deletes the original file after copying it. It's a lot like using `cp` and `rm` together. Let's take a look at what we can do:

```
/home/larry# cp /etc/passwd .
/home/larry# ls -F
passwd
/home/larry# mv passwd frog
/home/larry# ls -F
frog
/home/larry# mkdir report
/home/larry# mv frog report
/home/larry# ls -F
report/
/home/larry# ls -F report
frog
/home/larry#
```

As you can see, `mv` will rename a file if the second parameter is a file. If the second parameter is a directory, `mv` will move the file to the new directory, keeping it's shortname the same.

You should be very careful with `mv`—it doesn't check to see if the file already exists, and will remove any old file in its way. For instance, if I had a file named `frog` already in my directory `report`, the command `mv frog report` would delete the file `~/report/frog` and replace it with `~/frog`.

In fact, there is one way to make `rm`, `cp` and `mv` ask you before deleting files. All three of these commands accept the `-i` option, which makes them query the user before removing any file. If you use an **alias**, you can make the shell do `rm -i` automatically when you type `rm`. You'll learn more about this later in Section **??** on page **??**.

Chapter 2

The X Window System

The nice thing about standards is that there are so many of them to choose from.
Andrew S. Tanenbaum

This chapter only applies to those using the X Window System. If you encounter a screen with multiply windows, colors, or a cursor that is only movable with your mouse, you are using X. (If your screen consists of white characters on a black background, you are not currently using X. If you want to start it up, take a look at Section 2.1.)

2.1 Starting and Stopping the X Window System

2.1.1 Starting X

Even if X doesn't start automatically when you login, it is possible to start it from the regular text-mode shell prompt. There are two possible commands that will start X, either `startx` or `xinit`. Try `startx` first. If the shell complains that no such command is found, try using `xinit` and see if X starts. If neither command works, you may not have X installed on your system—consult local documentation for your distribution.

If the command runs but you are eventually returned to the black screen with the shell prompt, X is installed but not configured. Consult the documentation that came with your distribution on how to setup X.

2.1.2 Exiting X

Depending on how X is configured, there are two possible ways you might have to exit X. The first is if your window manager controls whether or not X is running. If it does, you'll have to exit X using a menu (see Section 2.4.8 on page 18). To display a menu, click a button on the background.

The important menu entry should be "Exit Window Manager" or "Exit X" or some entry containing the word "Exit". Try to find that entry (there could be more than one menu—try different mouse buttons!) and choose it.

The other method would be for a special `xterm` to control X. If this is the case, there is probably a window labeled "login" or "system xterm". To exit from X, move the mouse cursor into that window and type "exit".

If X was automatically started when you logged in, one of these methods should log you out. Simply login again to return. If you started X manually, these methods should return you to the text mode prompt. (If you wish to logout, type `logout` at this prompt.)

2.2 What is The X Window System?

The X Window System is a distributed, graphical method of working developed primarily at the Massachusetts Institute of Technology. It has since been passed to a consortium of vendors (aptly named "The X Consortium") and is being maintained by them.

The X Window System (hereafter abbreviated as "X"[1]) has new versions every few years, called releases. As of this writing, the latest revision is X11R6, or release six. The eleven in X11 is officially the version number but there hasn't been a new version in many years, and one is not currently planned.

There are two terms when dealing with X that you should be familiar. The **client** is a X program. For instance, `xterm` is the client that displays your shell when you log on. The **server** is a program that provides services to the client program. For instance, the server draws the window for `xterm` and communicates with the user.

Since the client and the server are two separate programs, it is possible to run the client and the server *on two physically separate machines*. In addition to supplying a standard method of doing graphics, you can run a program on a remote machine (across the country, if you like!) and have it display on the workstation right in front of you.

A third term you should be familiar with is the **window manager**. The window manager is a special client that tells the server where to position various windows and provides a way for the user to move these windows around. The server, by itself, does nothing for the user. It is merely there to provide a buffer between the user and the client.

2.3 What's This on my Screen?

When you first start X, several programs are started. First, the server is started. Then, several clients are usually started. Unfortunately, this is not standardized across various distributions. It is likely that among these clients are a window manager, either `fvwm` or `twm`, a prompt, `xterm`, and a clock, `xclock`.

2.3.1 XClock

`xclock` [-digital] [-analog] [-update *seconds*] [-hands *color*]

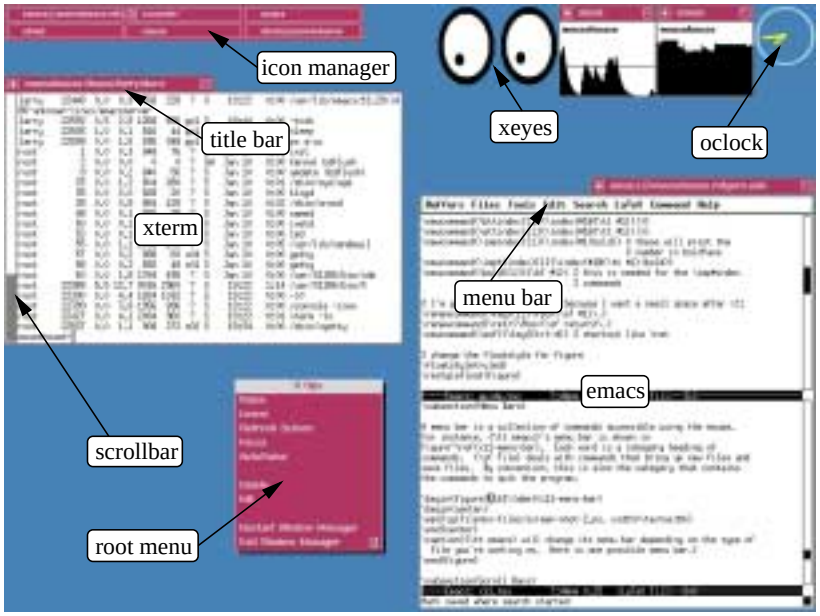
I'll explain the simplest one first: `xclock` functions exactly as you'd expect it would. It ticks off the seconds, minutes and hours in a small window.

No amounts of clicking or typing in `xclock`'s window will affect it—that's *all* it does. Or is it? In fact, there are various different options you can give to the program to have it act in different ways. For instance, `xclock -digital` will create a digital clock. `xclock -update 1` will create a second hand that moves every second, while `-update 5` will create a second hand that moves every 5 seconds.

For more information on `xclock`'s options, consult its manpage—`man xclock`. If you're going to try running a few of your own `xclock`s, you should probably read Section 3.4 (Multitasking) to learn how to run them in addition to your current programs. (If you run an `xclock` in the foreground—the usual way of running a program—and want to get out of it, type ctrl-c.)

[1] There are several acceptable ways to refer to The X Window System. A common though incorrect way of referring to X is "X Windows".

Figure 2.1 An annotated example of a standard X screen. In this example, the user is running `twm`. The standard clock has been replaced by a transparent clock called `oclock`.



2.3.2 XTerm

The window with a prompt in it (something that probably looks like `/home/larry#`) is being controlled by a program called `xterm`. `xterm` is a deceptively complicated program. At first glance, it doesn't seem to do much, but it actually has to do a lot of work. `xterm` emulates a terminal so that regular text-mode Unix applications work correctly. It also maintains a buffer of information so that you can refer back to old commands. (To see how to use this, look at Section 2.6.3.)

For much of this book, we're going to be learning about the Unix command-line, and you'll find that inside your `xterm` window. In order to type into `xterm`, you *usually* have to move your mouse cursor (possibly shaped like an "X" or an arrow) into the `xterm` window. However, this behavior is dependent on the window manager.

One way of starting more programs under X is through an `xterm`. Since X programs are standard Unix programs, they can be run from normal command prompts such as `xterm`s. Since running a long term program from a `xterm` would tie up the `xterm` as long as the program was running, people normally start X programs in the background. For more information about this, see Section 3.4.

2.4 Window Managers

On Linux, there are two different window managers that are commonly used. One of them, called `twm` is short for "Tab Window Manager". It is larger than the other window manager usually used,

`fvwm`. (`fvwm` stands for "F(?) Virtual Window Manager"—the author neglected to tie down exactly what the f stood for.) Both `twm` and `fvwm` are highly configurable, which means I can't tell you exactly what keys do what in your particular setup.

To learn about `twm`'s configuration, look at Section **??**. `fvwm`'s configuration is covered in Section **??**.

2.4.1 When New Windows are Created

There are three possible things a window manager will do when a new window is created. It is possible to configure a window manager so that an outline of the new window is shown, and you are allowed to position it on your screen. That is called **manual placement**. If you are presented with the outline of a window, simply use the mouse to place it where you wish it to appear and click the left mouse button.

It is also possible that the window manager will place the new window somewhere on the screen by itself. This is known as **random placement**.

Finally, sometimes an application will ask for a specific spot on the screen, or the window manager will be configured to display certain applications on the same place of the screen all the time. (For instance, I specify that I want `xclock` to always appear in the upper right hand corner of the screen.)

2.4.2 Focus

The window manager controls some important things. The first thing you'll be interested in is **focus**. The focus of the server is which window will get what you type into the keyboard. Usually in X the focus is determined by the position of the mouse cursor. If the mouse cursor is in one `xterm`'s window[2], that `xterm` will get your keypresses. This is different from many other windowing systems, such as Microsoft Windows, OS/2, or the Macintosh, where you must click the mouse in a window before that window gets focus. Usually under X, if your mouse cursor wanders from a window, focus will be lost and you'll no longer be able to type there.

Note, however, that it is possible to configure both `twm` and `fvwm` so that you must click on or in a window to gain focus, and click somewhere else to lose it, identical to the behavior of Microsoft Windows. Either discover how your window manager is configured by trial and error, or consult local documentation.

2.4.3 Moving Windows

Another very configurable thing in X is how to move windows around. In my personal configuration of `twm`, there are three different ways of moving windows around. The most obvious method is to move the mouse cursor onto the **title bar** and drag the window around the screen. Unfortunately, this may be done with any of the left, right, or middle buttons[3]. (To drag, move the cursor above the title bar, and hold down on the button while moving the mouse.) Most likely, your configuration is set to move windows using the *left* mouse buttons.

Another way of moving windows may be holding down a key while dragging the mouse. For instance, in *my* configuration, if I hold down the Alt key, move the cursor above a window, I can drag the window around using the left mouse button.

Again, you may be able to understand how the window manager is configured by trial and error, or by seeing local documentation. Alternatively, if you want to try to interpret the window manager's configuration file, see Section **??** for `twm` or Section **??** for `fvwm`.

[2] You can have more then one copy of `xterm` running at the same time!

[3] Many PCs have only two button mice. If this is the case for you, you should be able to emulate a middle button by using the left and right buttons simultaneously.

2.4.4 Depth

Since windows are allowed to overlap in X, there is a concept of **depth**. Even though the windows and the screen are both two dimensional, one window can be in front of another, partially or completely obscuring the rear window.

There are several operations that deal with depth:

- **Raising** the window, or bringing a window to the front. This is usually accomplished by clicking on a window's title bar with one of the buttons. Depending on how the window manager is configured, it could be any one of the buttons. (It is also possible that more then one button will do the job.)
- **Lowering** the window, or pushing the window to the back. This can generally be accomplished by a different click in the title bar. It is also possible to configure some window managers so that one click will bring the window foward if there is anything over it, while that same click will lower it when it is in the front.
- **Cycling** through windows is another operation many window managers allow. This brings each window to the front in an orderly cycle.

2.4.5 Iconization

There are several other operations that can obscure windows or hide them completely. First is the idea of "iconization". Depending on the window manager, this can be done in many different ways. In `twm`, many people configure an **icon manager**. This is a special window that contains a list of all the other windows on the screen. If you click on a name (depending on the setup, it could be with any of the buttons!) the window disappears—it is iconified. The window is still active, but you can't see it. Another click in the icon manager restores the window to the screen.

This is quite useful. For instance, you could have remote `xterm`s to many different computers that you occasionally use. However, since you rarely use all of them at a given time, you can keep most of the `xterm` windows iconified while you work with a small subset. The only problem with this is it becomes easy to "lose" windows. This causes you to create new windows that duplicate the functionality of iconified windows.

Other window managers might create actual icons across the bottom of the screen, or might just leave icons on the root window.

2.4.6 Resizing

There are several different methods to resize windows under X. Again, it is dependent on your window manager and exactly how your window manager is configured. The method many Microsoft Windows users are familiar with is to click on and drag the border of a window. If your window manager creates large borders that change how the mouse cursor looks when it is moved over them, that is probably the method used to resize windows.

Another method used is to create a "resizing" button on the titlebar. In Figure 2.3, a small button is visible on the right of each titlebar. To resize windows, the mouse is moved onto the resize button and the left mouse button is held down. You can then move the mouse outside the borders of the window to resize it. The button is released when the desired size has been reached.

2.4.7 Maximization

Most window managers support maximization. In `twm`, for instance, you can maximize the height, the width, or both dimensions of a window. This is called "zooming" in `twm`'s language although I prefer the term maximization. Different applications respond differently to changes in their window size. (For instance, `xterm` won't make the font bigger but will give you a larger workspace.)

Unfortunately, it is extremely non-standard on how to maximize windows.

2.4.8 Menus

Another purpose for window managers is for them to provide menus for the user to quickly accomplish tasks that are done over and over. For instance, I might make a menu choice that automatically launches Emacs or an additional `xterm` for me. That way I don't need to type in an `xterm`—an especially good thing if there aren't any running `xterm`s that I need to type in to start a new program!

In general, different menus can be accessed by clicking on the root window, which is an immovable window behind all the other ones. By default, it is colored gray, but could look like anything.[4] To try to see a menu, click and hold down a button on the desktop. A menu should pop up. To make a selection, move (without releasing the mouse button) the cursor over one of the items any then release the mouse button.

2.5 X Attributes

There are many programs that take advantage of X. Some programs, like `emacs`, can be run either as a text-mode program *or* as a program that creates its own X window. However, most X programs can only be run under X.

2.5.1 Geometry

There are a few things common to all programs running under X. In X, the concept of **geometry** is where and how large a window is. A window's geometry has four components:

- The horizontal size, usually measured in pixels. (A pixel is the smallest unit that can be colored. Many X setups on Intel PCs have 1024 pixels horizontally and 768 pixels vertically.) Some applications, like `xterm` and `emacs`, measure their size in terms of number of characters they can fit in the window. (For instance, eighty characters across.)
- The vertical size, also usually measured in pixels. It's possible for it to be measured in characters.
- The horizontal distance from one of the sides of the screen. For instance, `+35` would mean make the left edge of the window thirty-five pixels from the left edge of the screen. On the other hand, `-50` would mean make the right edge of the window fifty pixels from the right edge of the screen. It's generally impossible to start the window off the screen, although a window can be moved off the screen. (The main exception is when the window is very large.)
- The vertical distance from either the top or the bottom. A positive vertical distance is measured from the top of the screen; a negative vertical distance is measured from the bottom of the screen.

All four components get put together into a geometry string that looks like: `503x73-78+0`. (That translates into a window 503 pixels long, 73 pixels high, put near the top right hand corner of the screen.) Another way of stating it is *hsize*`x`*vsize*±*hplace*±*vplace*.

2.5.2 Display

Every X application has a display that it is associated with. The display is the name of the screen that the X server controls. A display consists of three components:

- The machine name that the server is running on. At stand-alone Linux installations the server is always running on the same system as the clients. In such cases, the machine name can be omitted.

[4] One fun program to try is called `xfishtank`. It places a small aquarium in the background for you.

Figure 2.2 Standard options for X programs.

Name	Followed by	Example
`-geometry`	geometry of the window	`xterm -geometry 80x24+0+90`
`-display`	display you want the program to appear	`xterm -display lionsden:0.0`
`-fg`	the primary foreground color	`xterm -fg yellow`
`-bg`	the primary background color	`xterm -bg blue`

- The number of the server running on that machine. Since any one machine could have multiple X servers running on it (unlikely for most LINUX machines, but possible) each must have a unique number.
- The screen number. X supports a particular server controlling more than one screen at a time. You can imagine that someone wants a lot of screen space, so they have two monitors sitting next to each other. Since they don't want two X servers running on one machine for performance reasons, they let one X server control both screens.

These three things are put together like so: *machine*:*server-number*.*screen-number*.

For instance, on `mousehouse`, all my applications have the display set to `:0.0`, which means the first screen of the first server on the local display. However, if I am using a remote computer, the display might be set to `mousehouse:0.0`.

By default, the display is taken from the environment variable (see Section **??**) named `DISPLAY`, and can be overridden with a command-line option (see Figure 2.2). To see how `DISPLAY` is set, try the command `echo $DISPLAY`.

2.6 Common Features

While X is a graphical user interface, it is a very uneven graphical user interface. It's impossible to say how any component of the system is going to work, because every component can easily be reconfigured, changed, and even replaced. This means it's hard to say exactly how to use various parts of the interface. We've already encountered one cause of this: the different window managers and how configurable each window manager is.

Another cause of this uneven interface is the fact that X applications are built using things called "widget sets". Included with the standard X distribution are "Athena widgets" developed at MIT. These are commonly used in free applications. They have the disadvantage that they are not particularly good-looking and are somewhat harder to use than other widgets.

The other popular widget set is called "Motif". Motif is a commercial widget set similar to the user interface used in Microsoft Windows. Many commercial applications use Motif widgets, as well as some free applications. The popular World Wide Web Browser `netscape` uses Motif.

Let's try to go through some of the more usually things you'll encounter.

2.6.1 Buttons

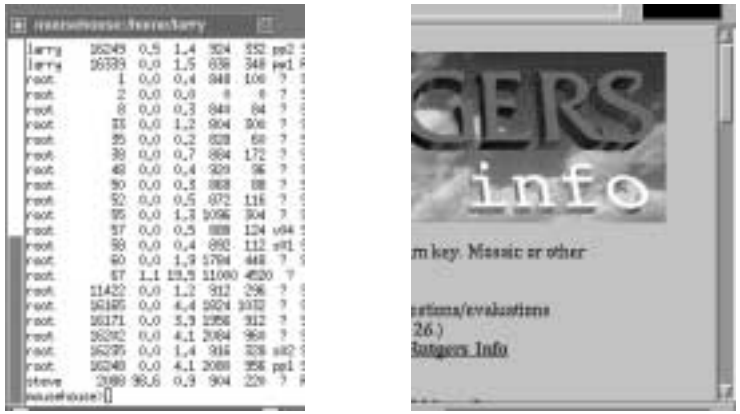
Buttons are generally the easiest thing to use. A button is invoked by positioning the mouse cursor over it and clicking (pressing and immediately releasing the mouse button) the left button. Athena and Motif buttons are functionally the same although they have cosmetic differences.

2.6.2 Menu Bars

A menu bar is a collection of commands accessible using the mouse. For instance, `emacs`'s menu bar is shown in Figure 2.3. Each word is a category heading of commands. File deals with commands

Figure 2.3 `emacs` will change its menu bar depending on the type of file you're working on. Here is one possible menu bar.

Figure 2.4 An Athena-type scroll bar is visible on the left of this `xterm` window. Next to it, a Motif-type scroll bar is visible on the `netscape` window.



that bring up new files and save files. By convention, this is also the category that contains the command to exit the program.

To access a command, move the mouse cursor over a particular category (such as File) and press and hold down the left mouse button. This will display a variety of commands. To select one of the commands, move the mouse cursor over that command and release the left mouse button. Some menu bars let you click on a category—if this is the case, clicking on the category will display the menu until you click on either a command, another menu, or outside the menu bar (indicating that you are not interested in running a particular command).

2.6.3 Scroll Bars

A **scroll bar** is a method to allow people to display only part of a document, while the rest is off the screen. For instance, the `xterm` window is currently displaying the bottom third of the text available in Figure 2.4. It's easy to see what part of the available text is current being displayed: the darkened part of the scroll bar is relative to both the position and the amount of displayed text. If the text displayed is all there is, the entire scroll bar is dark. If the middle half of the text is displayed, the middle half of the scroll bar is darkened.

A vertical scroll bar may be to the left or right of the text and a horizontal one may be above or below, depending the application.

2.6.3.1 Athena scroll bars

Athena scroll bars operate differently from scroll bars in other windowing systems. Each of the three buttons of the mouse operate differently. To scroll upwards (that is, display material above what is currently visible) you can click the rightmost mouse button anywhere in the scroll bar. To scroll downwards, click the left mouse button anywhere in the scroll bar.

You can also jump to a particular location in the displayed material by clicking the middle mouse button anywhere in the scroll bar. This causes the window to display material starting at that point in the document.

2.6.3.2 Motif scroll bars

A Motif scroll bar acts much more like a Microsoft Windows or Macintosh scroll bar. An example of one is on the right in Figure 2.4. Notice that in addition to the bar, it has arrows above and below it. These are used for fine-tuning: clicking either the left or middle buttons on them will scroll a small amount such as one line; the right button does nothing.

The behavior of clicking inside the scroll bar is widely different for Motif scroll bars than Athena scroll bars. The right button has no effect. Clicking the left button above the current position scrolls upward. Similarly, clicking below the current position scrolls downward. Clicking and holding the left button *on* the current position allows one to move the bar at will. Releasing the left button positions the window.

Clicking the middle button anywhere on the bar will immediately jump to that location, similar to the behavior of the Athena middle button. However, instead of starting to display the data at the position clicked, that position is taken to be the *midpoint* of the data to be displayed.

Chapter 3

Working with Unix

A UNIX saleslady, Lenore,
Enjoys work, but she likes the beach more.
She found a good way
To combine work and play:
She sells C shells by the seashore.

Unix is a powerful system for those who know how to harness its power. In this chapter, I'll try to describe various ways to use Unix's shell, `bash`, more efficently.

3.1 Wildcards

In the previous chapter, you learned about the file maintence commands `cp`, `mv`, and `rm`. Occasionally, you want to deal with more than one file at once—in fact, you might want to deal with many files at once. For instance, you might want to copy all the files beginning with `data` into a directory called `~/backup`. You could do this by either running many `cp` commands, or you could list every file on one command line. Both of these methods would take a long time, however, and you have a large chance of making an error.

A better way of doing that task is to type:

```
/home/larry/report# ls -F
1993-1          1994-1          data1           data5
1993-2          data-new        data2
/home/larry/report# mkdir ~/backup
/home/larry/report# cp data* ~/backup
/home/larry/report# ls -F ~/backup
data-new        data1           data2           data5
/home/larry/report#
```

As you can see, the asterix told `cp` to take all of the files beginning with `data` and copy them to `~/backup`. Can you guess what `cp d*w ~/backup` would have done?

3.1.1 What *Really* Happens?

Good question. Actually, there are a couple of special characters intercepted by the shell, `bash`. The character "`*`", an asterix, says "replace this word with all the files that will fit this specification". So, the command `cp data* ~/backup`, like the one above, gets changed to `cp data-new data1 data2 data5 ~/backup` before it gets run.

To illustrate this, let me introduce a new command, `echo`. `echo` is an extremely simple command; it echoes back, or prints out, any parameters. Thus:


```
/home/larry# echo Hello!
Hello!
/home/larry# echo How are you?
How are you?
/home/larry# cd report
/home/larry/report# ls -F
1993-1          1994-1          data1           data5
1993-2          data-new        data2
/home/larry/report# echo 199*
1993-1 1993-2 1994-1
/home/larry/report# echo *4*
1994-1
/home/larry/report# echo *2*
1993-2 data2
/home/larry/report#
```

As you can see, the shell expands the wildcard and passes all of the files to the program you tell it to run. This raises an interesting question: what happens if there are *no* files that meet the wildcard specification? Try `echo /rc/fr*og` and `bash` passes the wildcard specification verbatim to the program.

Other shells, like `tcsh`, will, instead of just passing the wildcard verbatim, will reply `No match.` Here's the same command run under `tcsh`:

```
mousehouse>echo /rc/fr*og
echo: No match.
mousehouse>
```

The last question you might want to know is what if I wanted to have `data*` echoed back at me, instead of the list of file names? Well, under both `bash` and `tcsh`, just include the string in quotes:

```
/home/larry/report# echo "data*"
data*
/home/larry/report#
```

OR

```
mousehouse>echo "data*"
data*
mousehouse>
```

3.1.2 The Question Mark

In addition to the asterix, the shell also interprets a question mark as a special character. A question mark will match one, and only one character. For instance, `ls /etc/??` will display all two letter files in the the `/etc` directory.

3.2 Time Saving with `bash`

3.2.1 Command-Line Editing

Occasionally, you've typed a long command to `bash` and, before you hit return, notice that there was a spelling mistake early in the line. You could just delete all the way back and retype everything you need to, but that takes much too much effort! Instead, you can use the arrow keys to move back there, delete the bad character or two, and type the correct information.

There are many special keys to help you edit your command line, most of them similar to the commands used in GNU Emacs. For instance, C-t flips two adjacent characters.[1] You'll be able to find most of the commands in the chapter on Emacs, Chapter 4.

[1] C-t means hold down the key labeled "Ctrl", then press the "t" key. Then release the "Ctrl" key.

3.2.2 Command and File Completion

Another feature of `bash` is automatic completion of your command lines. For instance, let's look at the following example of a typical `cp` command:

```
/home/larry# ls -F
this-is-a-long-file
/home/larry# cp this-is-a-long-file shorter
/home/larry# ls -F
shorter              this-is-a-long-file
/home/larry#
```

It's a big pain to have to type every letter of `this-is-a-long-file` whenever you try to access it. So, create `this-is-a-long-file` by copying `/etc/passwd` to it[2]. Now, we're going to do the above `cp` command very quickly and with a smaller chance of mistyping.

Instead of typing the whole filename, type `cp th` and press and release the Tab. Like magic, the rest of the filename shows up on the command line, and you can type in `shorter`. Unfortunately, `bash` cannot read your thoughts, and you'll have to type all of `shorter`.

When you type Tab, `bash` looks at what you've typed and looks for a file that starts like that. For instance, if I type `/usr/bin/ema` and then hit Tab, `bash` will find `/usr/bin/emacs` since that's the only file that begins `/usr/bin/ema` on my system. However, if I type `/usr/bin/ld` and hit Tab, `bash` beeps at me. That's because three files, `/usr/bin/ld`, `/usr/bin/ldd`, and `/usr/bin/ld86` all start with `/usr/bin/ld` on my system.

If you try a completion and `bash` beeps, you can immediately hit Tab again to get a list of all the files your start matches so far. That way, if you aren't sure of the exact spelling of your file, you can start it and scan a much smaller list of files.

3.3 The Standard Input and The Standard Output

Let's try to tackle a simple problem: getting a listing of the `/usr/bin` directory. If all we do is `ls /usr/bin`, some of the files scroll off the top of the screen. How can we see all of the files?

3.3.1 Unix Concepts

The Unix operating system makes it very easy for programs to use the terminal. When a program writes something to your screen, it is using something called **standard output**. Standard output, abbreviated as stdout, is how the program writes things to a user. The name for what you tell a program is **standard input** (stdin). It's possible for a program to communicate with the user without using standard input or output, but most of the commands I cover in this book use stdin and stdout.

For example, the `ls` command prints the list of the directories to standard output, which is normally "connected" to your terminal. An interactive command, such as your shell, `bash`, reads your commands from standard input.

It is also possible for a program to write to **standard error**, since it is very easy to make standard output point somewhere besides your terminal. Standard error (stderr) is almost always connected to a terminal so an actual human will read the message.

In this section, we're going to examine three ways of fiddling with the standard input and output: input redirection, output redirection, and pipes.

[2] `cp /etc/passwd this-is-a-long-file`

3.3.2 Output Redirection

A very important feature of Unix is the ability to **redirect** output. This allows you, instead of viewing the results of a command, to save it in a file or send it directly to a printer. For instance, to redirect the output of the command `ls /usr/bin`, we place a `>` sign at the end of the line, and say what file we want the output to be put in:

```
/home/larry# ls
/home/larry# ls -F /usr/bin > listing
/home/larry# ls
listing
/home/larry#
```

As you can see, instead of writing the names of all the files, the command created a totally new file in your home directory. Let's try to take a look at this file using the command `cat`. If you think back, you'll remember `cat` was a fairly useless command that copied what you typed (the standard input) to the terminal (the standard output). `cat` can also print a file to the standard output if you list the file as a parameter to `cat`:

```
/home/larry# cat listing
...
/home/larry#
```

The exact output of the command `ls /usr/bin` appeared in the contents of `listing`. All well and good, although it didn't solve the original problem.[3]

However, `cat` does do some interesting things when it's output is redirected. What does the command `cat listing > newfile` do? Normally, the `> newfile` says "take all the output of the command and put it in `newfile`." The output of the command `cat listing` is the file `listing`. So we've invented a new (and not so efficient) method of copying files.

How about the command `cat > fox`? `cat` by itself reads in each line typed at the terminal (standard input) and prints it right back out (standard output) until it reads Ctrl-d. In this case, standard output has been redirected into the file `fox`. Now `cat` is serving as a rudimentary editor:

```
/home/larry# cat > fox
The quick brown fox jumps over the lazy dog.
press Ctrl-d
```

We've now created the file `fox` that contains the sentence "The quick brown fox jumps over the lazy dog." One last use of the versitile `cat` command is to con**cat**enate files together. `cat` will print out every file it was given as a parameter, one after another. So the command `cat listing fox` will print out the directory listing of `/usr/bin`, and then it will print out our silly sentence. Thus, the command `cat listing fox > listandfox` will create a new file containing the contents of both `listing` and `fox`.

3.3.3 Input Redirection

Like redirecting standard output, it is also possible to redirect standard input. Instead of a program reading from your keyboard, it will read from a file. Since input redirection is related to output redirection, it seems natural to make the special character for input redirection be `<`. It too, is used after the command you wish to run.

This is generally useful if you have a data file and a command that expects input from standard input. Most commands also let you specify a file to operate on, so `<` isn't used as much in day-to-day operations as other techniques.

[3] For impatient readers, the command you might want to try is `more`. However, there's still a bit more to talk about before we get there.

3.3.4 The Pipe

Many Unix commands produce a large amount of information. For instance, it is not uncommon for a command like `ls /usr/bin` to produce more output than you can see on your screen. In order for you to be able to see all of the information that a command like `ls /usr/bin`, it's necessary to use another Unix command, called `more`.[4] `more` will pause once every screenful of information. For instance, `more < /etc/rc` will display the file `/etc/rc` just like `cat /etc/rc` would, except that `more` will let you read it. `more` also allows the command `more /etc/rc`, and that's the normal way of invoking it.

However, that doesn't help the problem that `ls /usr/bin` displays more information than you can see. `more < ls /usr/bin` won't work—input redirection only works with files, not commands! You *could* do this:

```
/home/larry# ls /usr/bin > temp-ls
/home/larry# more temp-ls
...
/home/larry# rm temp-ls
```

However, Unix supplies a much cleaner way of doing that. You can just use the command `ls /usr/bin | more`. The character "|" indicates a **pipe**. Like a water pipe, a Unix pipe controls flow. Instead of water, we're controlling the flow of information!

A useful tool with pipes are programs called **filters**. A filter is a program that reads the standard input, changes it in some way, and outputs to standard output. `more` is a filter—it reads the data that it gets from standard input and displays it to standard output one screen at a time, letting you read the file. `more` isn't a great filter because its output isn't suitable for sending to another program.

Other filters include the programs `cat`, `sort`, `head`, and `tail`. For instance, if you wanted to read only the first ten lines of the output from `ls`, you could use `ls /usr/bin | head`.

3.4 Multitasking

3.4.1 Using Job Control

Job control refers to the ability to put processes (another word for programs, essentially) in the **background** and bring them to the **foreground** again. That is to say, you want to be able to make something run while you go and do other things, but have it be there again when you want to tell it something or stop it. In Unix, the main tool for job control is the shell—it will keep track of jobs for you, if you learn how to speak its language.

The two most important words in that language are `fg`, for foreground, and `bg`, for background. To find out how they work, use the command `yes` at a prompt.

```
/home/larry# yes
```

This will have the startling effect of running a long column of `y`'s down the left hand side of your screen, faster than you can follow.[5] To get them to stop, you'd normally type ctrl-c to kill it, but instead you should type ctrl-z this time. It appears to have stopped, but there will be a message before your prompt, looking more or less like this:

```
[1]+   Stopped                 yes
```

[4] `more` is named because that's the prompt it originally displayed: `--more--`. In many versions of Linux the `more` command is identical to a more advanced command that does all that `more` can do and more. Proving that computer programmers make bad comedians, they named this new program `less`.

[5] There are good reasons for this strange command to exist. Occasional commands ask for confirmation—a "yes" answer to a question. The `yes` command allows a programmer to automate the response to these questions.

It means that the process `yes` has been **suspended** in the background. You can get it running again by typing `fg` at the prompt, which will put it into the foreground again. If you wish, you can do other things first, while it's suspended. Try a few `ls`'s or something before you put it back in the foreground.

Once it's returned to the foreground, the `y`'s will start coming again, as fast as before. You do not need to worry that while you had it suspended it was "storing up" more `y`'s to send to the screen: when a program is suspended the whole program doesn't run until you bring it back to life. (Now type ctrl-c to kill it for good, once you've seen enough).

Let's pick apart that message we got from the shell:

```
[1]+   Stopped                 yes
```

The number in brackets is the **job number** of this job, and will be used when we need to refer to it specifically. (Naturally, since job control is all about running multiple processes, we need some way to tell one from another). The `+` following it tells us that this is the "current job" — that is, the one most recently moved from the foreground to the background. If you were to type `fg`, you would put the job with the `+` in the foreground again. (More on that later, when we discuss running multiple jobs at once). The word `Stopped` means that the job is "stopped". The job isn't dead, but it isn't running right now. Linux has saved it in a special suspended state, ready to jump back into the action should anyone request it. Finally, the `yes` is the name of the process that has been stopped.

Before we go on, let's kill this job and start it again in a different way. The command is named `kill` and can be used in the following way:

```
/home/larry# kill %1
[1]+  Stopped                 yes
/home/larry#
```

That message about it being "stopped" again is misleading. To find out whether it's still alive (that is, either running or frozen in a suspended state), type `jobs`:

```
/home/larry# jobs
[1]+  Terminated                yes
/home/larry#
```

There you have it—the job has been terminated! (It's possible that the `jobs` command showed nothing at all, which just means that there are no jobs running in the background. If you just killed a job, and typing `jobs` shows nothing, then you know the kill was successful. Usually it will tell you the job was "terminated".)

Now, start `yes` running again, like this:

```
/home/larry# yes > /dev/null
```

If you read the section about input and output redirection, you know that this is sending the output of `yes` into the special file `/dev/null`. `/dev/null` is a black hole that eats any output sent to it (you can imagine that stream of `y`'s coming out the back of your computer and drilling a hole in the wall, if that makes you happy).

After typing this, you will not get your prompt back, but you will not see that column of `y`'s either. Although output is being sent into `/dev/null`, the job is still running in the foreground. As usual, you can suspend it by hitting ctrl-z. Do that now to get the prompt back.

```
/home/larry# yes > /dev/null
["yes" is running, and we just typed ctrl-z]
[1]+  Stopped                 yes >/dev/null

/home/larry#
```

Hmm... is there any way to get it to actually *run* in the background, while still leaving us the prompt for interactive work? The command to do that is `bg`:

```
/home/larry# bg
[1]+ yes >/dev/null  &
/home/larry#
```

Now, you'll have to trust me on this one: after you typed `bg`, `yes > /dev/null` began to run again, but this time in the background. In fact, if you do things at the prompt, like `ls` and stuff, you might notice that your machine has been slowed down a little bit (endlessly generating and discarding a steady stream of y's does take some work, after all!) Other than that, however, there are no effects. You can do anything you want at the prompt, and `yes` will happily continue to sending its output into the black hole.

There are now two different ways you can kill it: with the `kill` command you just learned, or by putting the job in the foreground again and hitting it with an interrupt, ctrl-c. Let's try the second way, just to understand the relationship between `fg` and `bg` a little better;

```
/home/larry# fg
yes >/dev/null

[now it's in the foreground again.  Imagine that I hit ctrl-c to terminate it]

/home/larry#
```

There, it's gone. Now, start up a few jobs running in simultaneously, like this:

```
/home/larry# yes  > /dev/null &
[1] 1024
/home/larry# yes | sort > /dev/null &
[2] 1026
/home/larry# yes | uniq > /dev/null
[and here, type ctrl-z to suspend it, please]

[3]+  Stopped                 yes | uniq >/dev/null
/home/larry#
```

The first thing you might notice about those commands is the trailing `&` at the end of the first two. Putting an `&` after a command tells the shell to start in running in the background right from the very beginning. (It's just a way to avoid having to start the program, type ctrl-z, and then type `bg`.) So, we started those two commands running in the background. The third is suspended and inactive at the moment. You may notice that the machine has become slower now, as the two running ones require some amount of CPU time.

Each one told you it's job number. The first two also showed you their **process identification numbers**, or PID's, immediately following the job number. The PID's are normally not something you need to know, but occasionally come in handy.

Let's kill the second one, since I think it's making your machine slow. You could just type `kill %2`, but that would be too easy. Instead, do this:

```
/home/larry# fg %2
yes | sort >/dev/null
[type ctrl-c to kill it]

/home/larry#
```

As this demonstrates, `fg` takes parameters beginning with `%` as well. In fact, you could just have typed this:


```
/home/larry# %2
yes | sort >/dev/null
[type ctrl-c to kill it]

/home/larry#
```

This works because the shell automatically interprets a job number as a request to put that job in the foreground. It can tell job numbers from other numbers by the preceding `%`. Now type `jobs` to see which jobs are left running:

```
/home/larry# jobs
[1]-  Running                 yes >/dev/null  &
[3]+  Stopped                 yes | uniq >/dev/null
/home/larry#
```

The "`-`" means that job number 1 is second in line to be put in the foreground, if you just type `fg` without giving it any parameters. The "`+`" means the specified job is first in line—a `fg` without parameters will bring job number 3 to the foreground. However, you can get to it by naming it, if you wish:

```
/home/larry# fg %1
yes >/dev/null
[now type ctrl-z to suspend it]

[1]+  Stopped                 yes >/dev/null
/home/larry#
```

Having changed to job number 1 and then suspending it has also changed the priorities of all your jobs. You can see this with the `jobs` command:

```
/home/larry# jobs
[1]+  Stopped                 yes >/dev/null
[3]-  Stopped                 yes | uniq >/dev/null
/home/larry#
```

Now they are both stopped (because both were suspended with ctrl-z), and number 1 is next in line to come to the foreground by default. This is because you put it in the foreground manually, and then suspended it. The "`+`" always refers to the most recent job that was suspended from the foreground. You can start it running again:

```
/home/larry# bg
[1]+ yes >/dev/null  &
/home/larry# jobs
[1]-  Running                 yes >/dev/null
[3]+  Stopped                 yes | uniq >/dev/null
/home/larry#
```

Notice that now it is running, and the other job has moved back up in line and has the `+`. Now let's kill them all so your system isn't permanently slowed by processes doing nothing.

```
/home/larry# kill %1 %3
[3]   Terminated              yes | uniq >/dev/null
/home/larry# jobs
[1]+  Terminated              yes >/dev/null
/home/larry#
```

You should see various messages about termination of jobs—nothing dies quietly, it seems. Figure 3.1 on the facing page shows a quick summary of what you should know for job control.

Figure 3.1 A summary of commands and keys used in job control.

`fg` *%job*	This is a shell command that returns a job to the foreground. To find out which one this is by default, type `jobs` and look for the one with the `+`. Parameters: Optional job number. The default is the process identified with `+`.
`&`	When an `&` is added to the end of the command line, it tells the command to run in the background automatically. This process is then subject to all the usual methods of job control detailed here.
`bg` *%job*	This is a shell command that causes a suspended job to run in the background. To find out which one this is by default, type `jobs` and look for the one with the `+`. Parameters: Optional job number. The default is the process identified with `+`.
`kill` *%job PID*	This is a shell command that causes a background job, either suspended or running, to terminate. You should always specify the job number or PID, and if you are using job numbers, remember to precede them with a `%`. Parameters: Either the job number (preceded by `%`) or PID (no `%` is necessary). More than one process or job can be specified on one line.
`jobs`	This shell command just lists information about the jobs currently running or suspending. Sometimes it also tells you about ones that have just exited or been terminated.
ctrl-c	This is the generic interrupt character. Usually, if you type it while a program is running in the foreground, it will kill the program (sometimes it takes a few tries). However, not all programs will respond to this method of termination.
ctrl-z	This key combination usually causes a program to suspend, although a few programs ignore it. Once suspended, the job can be run in the background or killed.

3.4.2 The Theory of Job Control

It is important to understand that job control is done by the shell. There is no program on the system called `fg`; rather, `fg`, `bg`, `&`, `jobs`, and `kill` are all shell-builtins (actually, sometimes `kill` is an independent program, but the `bash` shell used by Linux has it built in). This is a logical way to do it: since each user wants their own job control space, and each user already has their own shell, it is easiest to just have the shell keep track of the user's jobs. Therefore, each user's job numbers are meaningful only to that user: my job number [1] and your job number [1] are probably two totally different processes. In fact, if you are logged in more than once, each of your shells will have unique job control data, so you as a user might have two different jobs with the same number running in two different shells.

The way to tell for sure is to use the Process ID numbers (`PID`'s). These are system-wide — each process has its own unique `PID` number. Two different users can refer to a process by its `PID` and know that they are talking about the same process (assuming that they are logged into the same machine!)

Let's take a look at one more command to understand what `PID`s are. The `ps` command will list all running processes, including your shell. Try it out. It also has a few options, the most important of which (to many people) are `a`, `u`, and `x`. The `a` option will list processes belonging to any user, not just your own. The `x` switch will list processes that don't have a terminal associated with them.[6] Finally, the `u` switch will give additionally information about the process that is frequently useful.

[6] This only makes sense for certain system programs that don't have to talk to users through a keyboard.

To really get an idea of what your system is doing, put them all together: `ps -aux`. You can then see the process that uses the more memory by looking at the `%MEM` column, and the most CPU by looking at the `%CPU` column. (The `TIME` column lists the *total* amount of CPU time used.)

Another quick note about PIDs. `kill`, in addition to taking options of the form *%job#*, will take options of raw PIDs. So, put a `yes > /dev/null` in the background, run `ps`, and look for `yes`. Then type `kill PID`.[7]

If you start to program in C on your Linux system, you will soon learn that the shell's job control is just an interactive version of the function calls `fork` and `execl`. This is too complex to go into here, but may be helpful to remember later on when you are programming and want to run multiple processes from a single program.

3.5 Virtual Consoles: Being in Many Places at Once

Linux supports **virtual consoles**. These are a way of making your single machine seem like multiple terminals, all connected to one Linux kernel. Thankfully, using virtual consoles is one of the simplest things about Linux: there are "hot keys" for switching among the consoles quickly. To try it, log in to your Linux system, hold down the left Alt key, and press F2 (that is, the function key number 2).[8]

You should find yourself at another login prompt. Don't panic: you are now on virtual console (VC) number 2! Log in here and do some things — a few `ls`'s or whatever — to confirm that this is a real login shell. Now you can return to VC number 1, by holding down the left Alt and pressing F1. Or you can move on to a *third* VC, in the obvious way (Alt-F3).

Linux systems generally come with four VC's enabled by default. You can increase this all the way to eight; this should be covered in *The* Linux *System Adminstrator's Guide*. It involves editing a file in `/etc` or two. However, four should be enough for most people.

Once you get used to them, VC's will probably become an indispensable tool for getting many things done at once. For example, I typically run Emacs on VC 1 (and do most of my work there), while having a communications program up on VC 3 (so I can be downloading or uploading files by modem while I work, or running jobs on remote machines), and keep a shell up on VC 2 just in case I want to run something else without tying up VC 1.

[7] In general, it's easier to just kill the job number instead of using PIDs.

[8] Make sure you are doing this from text consoles: if you are running X windows or some other graphical application, it probably won't work, although rumor has it that X Windows will soon allow virtual console switching under Linux.

Chapter 4

Editing files with Emacs

FUNNY SOMETHING OR OTHER

4.1 What's Emacs?

In order to get anything done on a computer, you need a way to put text into files, and a way to change text that's already in files. An **editor** is a program for doing this. `Emacs` is one of the most popular editors around—partly because it's very easy for a complete beginner to get actual work done with it. (The classic Unix editor, `vi`, is covered in Appendix **??**.)

To learn `emacs`, you need to find a file of plain text (letters, numbers, and the like), copy it to your home directory[1] (we don't want to modify the actual file, if it contains important information), and invoke Emacs on the file:

```
/home/larry# emacs README
```

(Of course, if you decided to copy `/etc/rc`, `/etc/inittab`, or any other file, substitute that file name for `README`. For instance, if you `cp /etc/rc ~/rc`, then `emacs rc`.)

"Invoking" Emacs can have different effects depending on where where you do it. From a plain console displaying only text characters, Emacs will just take over the whole console. If you invoke it from X, Emacs will actually bring up its own window. I will assume that you are doing it from a text console, but everything carries over logically into the X Windows version—just substitute the word "window" in the places I've written "screen". Also, remeber that you have to move the mouse pointer into Emacs's window to type in it!

Your screen (or window, if you're using X) should now resemble Figure 4.1. Most of the screen contains your text document, but the last two lines are especially interesting if you're trying to learn Emacs. The second-to-last line (the one with the long string of dashes) is called the **mode line**.

In my mode line, you see "Top". It might be "All" instead, and there may be other minor differences. (Many people have the current time displayed in the mode line.) The line immediately below the mode line is called the **minibuffer**, or sometimes the **echo area**. Emacs uses the minibuffer to flash messages at you, and occasionally uses it to read input from you, when necessary. In fact, right now Emacs is telling you "`For information about the GNU Project and its goals, type C-h C-p.`" Ignore it for now; we won't be making much use of the minibuffer for a while.

Before you actually change any of the text in the file, you need to learn how to move around. The cursor should be at the beginning of the file, in the upper-left corner of the screen. To move forward, type `C-f` (that is, hold down the Control key while you press "f", for "forward"). It will move you forward a character at a time, and if you hold both keys down, your system's automatic key-repeat should take effect in a half-second or so. Notice how when you get to the end of the line,

[1] For instance, `cp /usr/src/linux/README ./README`

Figure 4.1 Emacs was just started with `emacs README`

```
        Linux kernel release 1.0

These are the release notes for linux version 1.0.  Read them carefully,
as they tell you what this is all about, explain how to install the
kernel, and what to do if something goes wrong.

WHAT IS LINUX?

  Linux is a Unix clone for 386/486-based PCs written from scratch by
  Linus Torvalds with assistance from a loosely-knit team of hackers
  across the Net.  It aims towards POSIX compliance.

  It has all the features you would expect in a modern fully-fledged
  Unix, including true multitasking, virtual memory, shared libraries,
  demand loading, shared copy-on-write executables, proper memory
  management and TCP/IP networking.

  It is distributed under the GNU General Public License - see the
  accompanying COPYING file for more details.

INSTALLING the kernel:
-----Emacs: README            (Fundamental)--Top-------------------------------
```

the cursor automatically moves to the next line. `C-b` (for "backward") has the opposite behavior. And, while we're at it, `C-n` and `C-p` take you to the next and previous lines, respectively.[2]

Using the control keys is usually the quickest way of moving around when you're editing. The goal of `Emacs` is to keep your hands over the alpha-numeric keys of the keyboard, where most of your work gets done. However, if you want to, the arrow keys should also work.

In fact, when you're using X, you should be able to position the mouse pointer and click with the left button to move the cursor where you want. However, this is very slow—you have to move your hand all the way to your mouse! Most people who use Emacs primarily use the keyboard for getting around.

Use `C-p` and `C-b` to get all the way back to the upper-left corner. Now keep `C-b` held a little longer. You should hear an annoying bell sound, and see the message "`Beginning of buffer`" appear in the minibuffer. At this point you might wonder, "But what is a buffer?"

When Emacs works on a file, it doesn't actually work on the file itself. Instead, it copies the contents of the file into a special Emacs work area called a **buffer**, where you can modify it to your heart's content. When you are done working, you tell Emacs to save the buffer—in other words, to write the buffer's contents into the corresponding file. Until you do this, the file remains unchanged, and the buffer's contents exist only inside of Emacs.

With that in mind, prepare to insert your first character into the buffer. Until now, everything we have done has been "non-destructive", so this is a big moment. You can choose any character you like, but if you want to do this in style, I suggest using a nice, solid, capital "X". As you type it, take a look at the beginning of the mode line at the bottom of the screen. When you change the buffer so that its contents are no longer the same as those of the file on disk, Emacs displays two asterisks at the beginning of the mode line, to let you know that the buffer has been modified:

```
--**-Emacs: some_file.txt           (Fundamental)--Top------------------------
```

These two asterisks are displayed as soon as you modify the buffer, and remain visible until you save the buffer. You can save the buffer multiple times during an editing session—the command to do so is just `C-x C-s` (hold down Control and hit "x" and "s" while it's down... okay, so you

[2] In case you hadn't noticed yet, many of Emacs's movement commands consist of combining Control with a single mnemonic letter.

probably already figured that out!). It's deliberately easy to type, because saving your buffers is something best done early and often.

I'm going to list a few more commands now, along with the ones you've learned already, and you can practice them however you like. I'd suggest becoming familiar with them before going any further:

`C-f`	Move forward one character.
`C-b`	Move backward one character.
`C-n`	Go to next line.
`C-p`	Go to previous line.
`C-a`	Go to beginning of line.
`C-e`	Go to end of line.
`C-v`	Go to next page/screenful of text.
`C-l`	Redraw the screen, with current line in center.
`C-d`	Delete this character (practice this one).
`C-k`	Delete text from here to end of line.
`C-x C-s`	Save the buffer in its corresponding file.
Backspace	Delete preceding character (the one you just typed).

4.2 Getting Started Quickly in X

If all you're interesting in is editing a few files quickly, an X user doesn't have to go much further beyond the menus at the top of the screen:

These menus are not available in text mode.

When you first start Emacs, there will be four menus at the top of the screen: Buffers, File, Edit, and Help. To use a menu, simply move the mouse pointer over the name (like File, click and hold down on the left button. Then, move the pointer to the action you want and release the mouse button. If you change your mind, move the mouse pointer away from the menu and release the button.

The Buffers menu lists the different files you've been editing in this incarnation of Emacs.The File menu shows a bunch of commands for loading and saving files—many of them will be described later. The Edit menu displays some commands for editing one buffer, and the Help menu should hopefully give on-line documentation.

You'll notice keyboard equivalents are listed next to the choices in the menu. Since, in the long run, they'll be quicker, you might want to learn them. Also, for better or for worse, most of Emacs's functionality is *only* available through the keyboard—you might want to read the rest of this chapter.

4.3 Editing Many Files at Once

Emacs can work on more than one file at a time. In fact, the only limit on how many buffers your Emacs can contain is the actual amount of memory available on the machine. The command to bring a new file into an Emacs buffer is `C-x C-f`. When you type it, you will be prompted for a filename in the minibuffer:

```
Find file: ~/
```

The syntax here is the same one used to specify files from the shell prompt; slashes represent subdirectories, ~means your home directory. You also get **filename completion**, meaning that if

you've typed enough of a filename at the prompt to identify the file uniquely, you can just hit Tab to complete it (or to show possible completions, if there are more than one). Space also has a role in filename completion in the minibuffer, similar to Tab, but I'll let you experiment to find out how the two differ. Once you have the full filename in the minibuffer, hit Return, and Emacs will bring up a buffer displaying that file. In Emacs, this process is known as **finding** a file. Go ahead and find some other unimportant text file now, and bring it into Emacs (do this from our original buffer `some_file.txt`). Now you have a new buffer; I'll pretend it's called `another_file.txt`, since I can't see your mode line.

Your original buffer seems to have disappeared—you're probably wondering where it went. It's still inside Emacs, and you can switch back to it with `C-x b`. When you type this, you will see that the minibuffer prompts you for a buffer to switch to, and it names a default. The default is the buffer you'd get if you just hit Return at the prompt, without typing a buffer name. The default buffer to switch to is always the one most recently left, so that when you are doing a lot of work between two buffers, `C-x b` always defaults to the "other" buffer (which saves you from having to type the buffer name). Even if the default buffer is the one you want, however, you should try typing in its name anyway.

Notice that you get the same sort of completion you got when finding a file: hitting Tab completes as much of a buffer name as it can, and so on. Whenever you are being prompted for something in the minibuffer, it's a good idea to see if Emacs is doing completion. Taking advantage of completion whenever it's offered will save you a lot of typing. Emacs usually does completion when you are choosing one item out of some predefined list.

Everything you learned about moving around and editing text in the first buffer applies to the new one. Go ahead and change some text in the new buffer, but don't save it (i.e. don't type `C-x C-s`). Let's assume that you want to discard your changes without saving them in the file. The command for that is `C-x k`, which "kills" the buffer. Type it now. First you will be asked which buffer to kill, but the default is the current buffer, and that's almost always the one you want to kill, so just hit Return. Then you will be asked if you *really* want to kill the buffer—Emacs always checks before killing a buffer that has unsaved changes in it. Just type "yes" and hit Return, if you want to kill it.

Go ahead and practice loading in files, modifying them, saving them, and killing their buffers. Make sure you don't modify any important system files in a way that will cause trouble[3], of course, but do try to have at least five buffers open at once, so you can get the hang of switching between them.

4.4 Ending an Editing Session

When you are done with your work in Emacs, make sure that all buffers are saved that should be saved, and exit Emacs with `C-x C-c`. Sometimes `C-x C-c` will ask you a question or two in the minibuffer before it lets you leave—don't be alarmed, just answer them in the obvious ways. If you think that you might be returning to Emacs later, don't use `C-x C-c` at all; use `C-z`, which will suspend Emacs. You can return to it with the shell command "`fg`" later. This is more efficient than stopping and starting Emacs multiple times, especially if you have edit the same files again later.

Under X, hitting `C-z` will merely iconize the window. See the section on iconization in Chapter 2. This gives you two ways of iconizing Emacs—the normal way your window manager offers, and `C-z`. Remember, when you iconize, a simply `fg` won't bring the window back—you'll have to use your window manager.

[3] If you are not the "root" user on the machine, you shouldn't be able to hurt the system anyway, but be careful just the same.

4.5 The Meta Key

You've already learned about one "modifier key" in Emacs, the Control key. There is a second one, called the **Meta** key, which is used almost as frequently. However, not all keyboards have their Meta key in the same place, and some don't have one at all. The first thing you need to do is find where your Meta key is located. Chances are, your keyboard's Alt keys are also Meta keys, if you are using an IBM PC or other another keyboard that has an Alt key.

The way to test this is to hold down a key that you think might be a Meta key and type "x". If you see a little prompt appear in the minibuffer (like this: `M-x`) then you've found it. To get rid of the prompt and go back to your Emacs buffer, type `C-g`.

If you didn't get a prompt, then there is still one solution. You can use the Escape key as a Meta key. But instead of holding it down while you type the next letter, you have to tap it and release it quickly, and *then* type the letter. This method will work whether or not you have a real Meta key, so it's the safest way to go. Try tapping Escape and then typing "x" now. You should get that tiny prompt again. Just use `C-g` to make it go away. `C-g` is the general way in Emacs to quit out of something you don't mean to be in. It usually beeps annoyingly at you to let you know that you have interrupted something, but that's fine, since that's what you intended to do if you typed `C-g`![4]

The notation `M-x` is analogous to `C-x` (substitute any character for "`x`"). If you have found a real Meta key, use that, otherwise just use the Escape key. I will simply write `M-x` and you'll have to use your own Meta key.

4.6 Cutting, Pasting, Killing and Yanking

Emacs, like any good editor, allows you to cut and paste blocks of text. In order to do this, you need a way to define the start and end of the block. In Emacs, you do this by setting two locations in the buffer, known as **mark** and **point**. To set the mark, go to the place you want your block to begin and type `C-SPC` ("`SPC`" means Space, of course). You should see the message "Mark set" appear in the minibuffer.[5] The mark has now been set at that place. There will be no special highlighting indicating that fact, but you know where you put it, and that's all that matters.

What about **point**? Well, it turns out that you've been setting point every time you move the cursor, because "point" just refers to your current location in the buffer. In formal terms, point is the spot where text would be inserted if you were to type something. By setting the mark, and then moving to the end of the block of text, you have actually defined a block of text. This block is known as the **region**. The region always means the area between mark and point.

Merely defining the region does not make it available for pasting. You have to tell Emacs to copy it in order to be able to paste it. To copy the region, make sure that mark and point are set correctly, and type `M-w`. It has now been recorded by Emacs. In order to paste it somewhere else, just go there and type `C-y`. This is known as **yanking** the text into the buffer.

If you want to actually move the text of the region to somewhere else, type `C-w` instead of `M-w`. This will **kill** the region—all the text inside it will disappear. In fact, it has been saved in the same way as if you had used `M-w`. You can yank it back out with `C-y`, as always. The place Emacs saves all this text is known as the **kill-ring**. Some editors call it the "clipboard" or the "paste buffer".

There's another way to do cutting and pasting: whenever you use `C-k` to kill to the end of a line, the killed text is saved in the kill-ring. If you kill more than one line in a row, they are all saved in the kill-ring together, so that the next yank will paste in all the lines at once. Because of this feature, it is often faster to use repeated `C-k`'s to kill some text than it is to explicitly set mark and point and use `C-w`. However, either way will work. It's really a matter of personal preference how you do it.

[4] Occasionally, even one `C-g` isn't enough to persuade Emacs that you really wanted to interrupt what you're doing. Just keep at it, and Emacs will usually return to a saner mode.

[5] On some terminals, `C-SPC` doesn't work. For these machines, you must use `C-@`.

4.7 Searching and Replacing

There are several ways to search for text in Emacs. Many of them are rather complex, and not worth going into here. The easiest and most entertaining way is to use **isearch**. "Isearch" stands for "incremental search". Suppose you want to search for the string "gadfly" in the following buffer:

```
I was growing afraid that we would run out of gasoline, when my passenger exclaime
``Gadzooks!  There's a gadfly in here!''.
```

You would move to the beginning of the buffer, or at least to some point that you know is before the first occurence of the goal word, "gadfly", and type `C-s`. That puts you in isearch mode. Now start typing the word you are searching for, "gadfly". But as soon as you type the "g", you see that Emacs has jumped you to the first occurence of "g" in the buffer. If the above quote is the entire contents of the buffer, then that would be the first "g" of the word "growing". Now type the "a" of "gadfly", and Emacs leaps over to "gasoline", which contains the first occurence of a "ga". The "d" gets you to gadzooks, and finally, "f" gets you to "gadfly", without your having had to type the entire word.

What you are doing in an isearch is defining a string to search for. Each time you add a character to the end of the string, the number of matches is reduced, until eventually you have entered enough to define the string uniquely. Once you have found the match you are looking for, you can exit the search with Return or any of the normal movement commands. If you think the string you're looking for is behind you in the buffer, then you should use `C-r`, which does an isearch backwards.

If you encounter a match, but it's not the one you were looking for, then hit `C-s` again while still in the search. This will move you forward to the next complete match, each time you hit it. If there is no next match, it will say that the search failed, but if you press `C-s` again at that point, the search will wrap around from the beginning of the buffer. The reverse holds true for `C-r` — it wraps around the end of the buffer.

Try bringing up a buffer of plain English text and doing and isearch for the string "`the`". First you'd type in as much as you wanted, then use repeated `C-s`'s to go to all instances of it. Notice that it will match words like "`them`" as well, since that also contains the substring "`the`". To search only for "`the`", you'd have to do add a space to the end of your search string. You can add new characters to the string at any point in the search, even after you've hit `C-s` repeatedly to find the next matches. You can also use Backspace or Delete to remove characters from the search string at any point in the search, and hitting Return exits the search, leaving you at the last match.

Emacs also allows you to replace all instances of a string with some new string—this is known as **query-replace**. To invoke it, type `query-replace` and hit Return. Completion is done on the command name, so once you have typed "query-re", you can just hit Tab to finish it. Say you wish to replace all instances of "gadfly" with "housefly". At the "`Query replace:  `" prompt, type "gadfly", and hit Return. Then you will be prompted again, and you should enter "housefly". Emacs will then step through the buffer, stopping at every instance of the word "gadfly", and asking if you want to replace it. Just hit "`y`" or "`n`" at each instance, for "Yes" or "No", until it finishes. If this doesn't make sense as you read it, then try it out.

4.8 What's Really Going On Here?

Actually, all these **keybindings** you have been learning are shortcuts to Emacs functions. For example, `C-p` is a short way of telling Emacs to execute the internal function `previous-line`. However,

all these internal functions can be called by name, using `M-x`. If you forgot that `previous-line` is bound to `C-p`, you could just type `M-x previous-line` Return, and it would move you up one line. Try this now, to understand how `M-x previous-line` and `C-p` are really the same thing.

The designer of Emacs started from the ground up, first defining a whole lot of internal functions, and then giving keybindings to the most commonly-used ones. Sometimes it's easier just to call a function explicitly with `M-x` than to remember what key it's bound to. The function `query-replace`, for example, is bound to `M-%` in some versions of Emacs. But who can remember such an odd keybinding? Unless you use query-replace extremely often, it's easier just to call it with `M-x`.

Most of the keys you type are letters, meant to be inserted into the text of the buffer. So each of those keys is **bound** to the function `self-insert-command`, which does nothing but insert that letter into the buffer. Combinations that use the Control key with a letter are generally bound to functions that do other things, like moving you around. For example, `C-v` is bound to a function called `scroll-up`, which scrolls the buffer up by one screenful (meaning that your position in the buffer moves *down*, of course).

If you ever actually wanted to insert a Control character into the buffer, then, how would you do it? After all, the Control characters are `ASCII` characters, although rarely used, and you might want them in a file. There is a way to prevent Control characters from being interpreted as commands by Emacs. The key `C-q`[6] is bound to a special function named `quoted-insert`. All `quoted-insert` does is read the next key and insert it literally into the buffer, without trying to interpret it as a command. This is how you can put Control characters into your files using Emacs. Naturally, the way to insert a C-q is to press `C-q` twice!

Emacs also has many functions that are not bound to any key. For example, if you're typing a long message, you don't want to have to hit return at the end of every line. You can have Emacs do it for you (you can have Emacs do anything for you)—the command to do so is called `auto-fill-mode`, but it's not bound to any keys by default. In order to invoke this command, you would type "`M-x auto-fill-mode`". "`M-x`" is the key used to call functions by name. You could even use it to call functions like `next-line` and `previous-line`, but that would be very inefficient, since they are already bound to `C-n` and `C-p`!

By the way, if you look at your mode line after invoking `auto-fill-mode`, you will notice that the word "Fill" has been added to the right side. As long as it's there, Emacs will fill (wrap) text automatically. You can turn it off by typing "`M-x auto-fill-mode`" again—it's a toggle command.

The inconvenience of typing long function names in the minibuffer is lessened because Emacs does completion on function names the same way it does on file names. Therefore, you should rarely find yourself typing in the whole function name, letter by letter. If you're not sure whether or not you can use completion, just hit Tab. It can't hurt: the worst thing that will happen is that you'll just get a tab character, and if you're lucky, it'll turn out that you can use completion.

4.9 Asking Emacs for Help

Emacs has extensive help facilities—so extensive, in fact, that we can only touch on them here. The most basic help features are accessed by typing `C-h` and then a single letter. For example, `C-h k` gets help on a key (it prompts you to type a key, then tells you what that key does). `C-h t` brings up a short Emacs tutorial. Most importantly, `C-h C-h C-h` gets you help on help, to tell you what's available once you have typed `C-h` the first time. If you know the name of an Emacs function (`save-buffer`, for example), but can't remember what key sequence invokes it, then use `C-h w`, for "`where-is`", and type in the name of the function. Or, if you want to know what a function does in detail, use `C-h f`, which prompts for a function name.

Remember, since Emacs does completion on function names, you don't really have to be sure what a function is called to ask for help on it. If you think you can guess the word it might start

[6] We call `C-q` a "key", even though it is produced by holding down Control and pressing "q", because it is a single `ASCII` character.

with, type that and hit Tab to see if it completes to anything. If not, back up and try something else. The same goes for file names: even if you can't remember quite what you named some file that you haven't accessed for three months, you can guess and use completion to find out if you're right. Get used to using completion as means of asking questions, not just as a way of saving keystrokes.

There are other characters you can type after `C-h`, and each one gets you help in a different way. The ones you will use most often are `C-h k`, `C-h w`, and `C-h f`. Once you are more familiar with Emacs, another one to try is `C-h a`, which prompts you for a string and then tells you about all the functions who have that string as part of their name (the "a" means for "apropos", or "about").

Another source of information is the **Info** documentation reader. Info is too complex a subject to go into here, but if you are interested in exploring it on your own, type `C-h i` and read the paragraph at the top of the screen. It will tell you how get more help.

4.10 Specializing Buffers: Modes

Emacs buffers have **modes** associated with them[7]. The reason for this is that your needs when writing a mail message are very different from your needs when, say, writing a program. Rather than try to come up with an editor that would meet every single need all the time (which would be impossible), the designer of Emacs[8] chose to have Emacs behave differently depending on what you are doing in each individual buffer. Thus, buffers have modes, each one designed for some specific activity. The main features that distinguish one mode from another are the keybindings, but there can be other differences as well.

The most basic mode is `fundamental` mode, which doesn't really have any special commands at all. In fact, here's what Emacs has to say about Fundamental Mode:

```
Fundamental Mode:

Major mode not specialized for anything in particular.
Other major modes are defined by comparison with this one.
```

I got that information like this: I typed `C-x b`, which is `switch-to-buffer`, and entered "foo" when it prompted me for a buffer name to switch to. Since there was previously no buffer named "`foo`", Emacs created one and switched me to it. It was in `fundamental-mode` by default, but it it hadn't been, I could have typed "`M-x fundamental-mode`" to make it so. All mode names have a command called `<modename>-mode` which puts the current buffer into that mode. Then, to find out more information about that major mode, I typed `C-h m`, which gets you help on the current major mode of the buffer you're in.

There's a slightly more useful mode called `text-mode`, which has the special commands `M-S`, for `center-paragraph`, and `M-s`, which invokes `center-line`. `M-S`, by the way, means exactly what you think it does: hold down both the Meta and the Shift key, and press "S".

Don't just take my word for this—go make a new buffer, put it into `text-mode`, and type `C-h m`. You may not understand everything Emacs tells you when you do that, but you should be able to get some useful information out of it.

Here is an introduction to some of the more commonly used modes. If you use them, make sure that you type `C-h m` sometime in each one, to find out more about each mode.

[7] To make matters worse, there are "Major Modes" and "Minor Modes", but you don't need to know about that right now.

[8] Richard Stallman, also sometimes referred to as "`rms`", because that's his login name.

4.11 Programming Modes

4.11.1 C Mode

If you use Emacs for programming in the C language, you can get it to do all the indentation for you automatically. Files whose names end in ".c " or ".h" are automatically brought up in `c-mode`. This means that certain special editing commands, useful for writing C-programs, are available. In C-mode, Tab is bound to `c-indent-command`. This means that hitting the Tab key does not actually insert a tab character. Instead, if you hit Tab anywhere on a line, Emacs automatically indents that line correctly for its location in the program. This implies that Emacs knows something about C syntax, which it does (although nothing about semantics—it cannot insure that your program has no errors!)

In order to do this, it assumes that the previous lines are indented correctly. That means that if the preceding line is missing a parenthesis, semicolon, curly brace, or whatever, Emacs will indent the current line in a funny way. When you see it do that, you will know to look for a punctuation mistake on the line above.

You can use this feature to check that you have punctuated your programs correctly—instead of reading through the entire program looking for problems, just start indenting lines from the top down with Tab, and when something indents oddly, check the lines just before it. In other words, let Emacs do the work for you!

4.11.2 Scheme Mode

This is a major mode that won't do you any good unless you have a compiler or an interpreter for the Scheme programming language on your system. Having one is not as normal as having, say, a C compiler, but it's becoming more and more common, so I'll cover it too. Much of what is true for Scheme mode is true for Lisp mode as well, if you prefer to write in Lisp.

Well, to make matters painful, Emacs comes with two different Scheme modes, because people couldn't decide how they wanted it to work. The one I'm describing is called `cmuscheme`, and later on, in the section on customizing Emacs, I'll talk about how there can be two different Scheme modes and what to do about it. For now, don't worry about it if things in your Emacs don't quite match up to what I say here. A customizable editor means an unpredictable editor, and there's no way around that!

You can run an interactive Scheme process in Emacs, with the command `M-x run-scheme`. This creates a buffer named "`*scheme*`", which has the usual Scheme prompt in it. You can type in Scheme expressions at the prompt, hit Return, and Scheme will evaluate them and display the answer. Thus, in order to interact with the Scheme process, you could just type all your function definitions and applications in at the prompt. Chances are you have previously-written Scheme source code in a file somewhere, and it would be easier to do your work in that file and send the definitions over to the Scheme process buffer as necessary.

If that source file ends in "`.ss`" or "`.scm`", it will automatically be brought up in **Scheme mode** when you find it with `C-x C-f`. If for some reason, it doesn't come up in Scheme mode, you can do it by hand with `M-x scheme-mode`. This `scheme-mode` is not the same thing as the buffer running the Scheme process; rather, the source code buffer's being in `scheme-mode` means that it has special commands for communicating with the process buffer.

If you put yourself inside a function definition in the Scheme source code buffer and type `C-c C-e`, then that definition will be "sent" to the process buffer — exactly as if you had typed it in yourself. `C-c M-e` sends the definition and then brings you to the process buffer to do some interactive work. `C-c C-l` loads a file of Scheme code (this one works from either the process buffer or the source code buffer). And like other programming language modes, hitting Tab anywhere on a line of code correctly indents that line.

If you're at the prompt in the process buffer, you can use `M-p` and `M-n` to move through your previous commands (also known as the **input history**). So if you are debugging the function '`rotate`', and have already applied it to arguments in the process buffer, like so:

```
> (rotate '(a b c d e))
```

then you can get that command back by typing `M-p` at the prompt later on. There should be no need to retype long expressions at the Scheme prompt — get in the habit of using the input history and you'll save a lot of time.

Emacs knows about quite a few programming languages: C, C++, Lisp, and Scheme are just some. Generally, it knows how to indent them in intuitive ways.

4.11.3 Mail Mode

You can also edit and send mail in Emacs. To enter a mail buffer, type `C-x m`. You need to fill in the `To:` and `Subject:` fields, and then use C-n to get down below the separator line into the body of the message (which is empty when you first start out). Don't change or delete the separator line, or else Emacs will not be able to send your mail—it uses that line to distinguish the mail's headers, which tell it where to send the mail, from the actual contents of the message.

You can type whatever you want below the separator line. When you are ready to send the message, just type `C-c C-c`, and Emacs will send it and then make the mail buffer go away.

4.12 Being Even More Efficient

Experienced Emacs users are fanatical about efficiency. In fact, they will often end up wasting a lot of time searching for ways to be more efficient! While I don't want that to happen to you, there are some easy things you can do to become a better Emacs user. Sometimes experienced users make novices feel silly for not knowing all these tricks—for some reason, people become religious about using Emacs "correctly". I'd condemn that sort of elitism more if I weren't about to be guilty of it myself. Here we go:

When you're moving around, use the fastest means available. You know that `C-f` is `forward-char`—can you guess that `M-f` is `forward-word`? `C-b` is `backward-char`. Guess what `M-b` does? That's not all, though: you can move forward a sentence at a time with `M-e`, as long as you write your sentences so that there are always two spaces following the final period (otherwise Emacs can't tell where one sentence ends and the next one begins). `M-a` is `backward-sentence`.

If you find yourself using repeated `C-f`'s to get to the end of the line, be ashamed, and make sure that you use `C-e` instead, and `C-a` to go to the beginning of the line. If you use many `C-n`'s to move down screenfuls of text, be very ashamed, and use `C-v` forever after. If you are using repeated `C-p`'s to move up screenfuls, be embarrassed to show your face, and use `M-v` instead.

If you are nearing the end of a line and you realize that there's a mispelling or a word left out somewhere earlier in the line, *don't* use Backspace or Delete to get back to that spot. That would require retyping whole portions of perfectly good text. Instead, use combinations of `M-b`, `C-b`, and `C-f` to move to the precise location of the error, fix it, and then use `C-e` to move to the end of the line again.

When you have to type in a filename, don't ever type in the whole name. Just type in enough of it to identify it uniquely, and let Emacs's completion finish the job by hitting Tab or Space. Why waste keystrokes when you can waste CPU cycles instead?

If you are typing some kind of plain text, and somehow your auto-filling (or auto-wrapping) has gotten screwed up, use `M-q`, which is `fill-paragraph` in common text modes. This will "adjust" the paragraph you're in as if it had been wrapped line by line, but without your having to go mess around with it by hand. `M-q` will work from inside the paragraph, or from its very beginning or end.

Sometimes it's helpful to use `C-x u`, (**undo**), which will try to "undo" the last change(s) you made. Emacs will guess at how much to undo; usually it guesses very intelligently. Calling it repeatedly will undo more and more, until Emacs can no longer remember what changes were made.

4.13 Customizing Emacs

Emacs is *so* big, and *so* complex, that it actually has its own programming language! I'm not kidding: to really customize Emacs to suit your needs, you have to write programs in this language. It's called Emacs Lisp, and it's a dialect of Lisp, so if you have previous experience in Lisp, it will seem quite friendly. If not, don't worry: I'm not going to go into a great deal of depth, because it's definitely best learned by doing. To really learn about programming Emacs, you should consult the Info pages on Emacs Lisp, and read a lot of Emacs Lisp source code.

Most of Emacs's functionality is defined in files of Emacs Lisp[9] code. Most of these files are distributed with Emacs and collectively are known as the "Emacs Lisp library". This library's location depends on how Emacs was installed on your system — common locations are `/usr/lib/emacs/lisp`, `/usr/lib/emacs/19.19/lisp/`, etc. The "`19.19`" is the version number of Emacs, and might be different on your system.

You don't need to poke around your filesystem looking for the lisp library, because Emacs has the information stored internally, in a variable called **load-path**. To find out the value of this variable, it is necessary to **evaluate** it; that is, to have Emacs's lisp interpreter get its value. There is a special mode for evaluating Lisp expressions in Emacs, called **lisp-interaction-mode**. Usually, there is a buffer called "`*scratch*`" that is already in this mode. If you can't find one, create a new buffer of any name, and type `M-x lisp-interaction-mode` inside it.

Now you have a workspace for interacting with the Emacs Lisp interpreter. Type this:

```
load-path
```

and then press `C-j` at the end of it. In lisp-interaction-mode, `C-j` is bound to `eval-print-last-sexp`. An "`sexp`" is an "**s-expression**", which means a balanced group of parentheses, including none. Well, that's simplifying it a little, but you'll get a feel for what they are as you work with Emacs Lisp. Anyway, evaluating `load-path` should get you something like this:

```
load-path C-j
("/usr/lib/emacs/site-lisp/vm-5.35" "/home/kfogel/elithp"
"/usr/lib/emacs/site-lisp" "/usr/lib/emacs/19.19/lisp")
```

It won't look the same on every system, of course, since it is dependant on how Emacs was installed. The above example comes from my 386 PC running Linux. As the above indicates, `load-path` is a list of strings. Each string names a directory that might contain Emacs Lisp files. When Emacs needs to load a file of Lisp code, it goes looking for it in each of these directories, in order. If a directory is named but does not actually exist on the filesystem, Emacs just ignores it.

When Emacs starts up, it automatically tries to load the file `.emacs` in your home directory. Therefore, if you want to make personal customizations to Emacs, you should put them in `.emacs`. The most common customizations are keybindings, so here's how to do them:

```
(global-set-key "\C-cl" 'goto-line)
```

`global-set-key` is a function of two arguments: the key to be bound, and the function to bind it to. The word "`global`" means that this keybinding will be in effect in all major modes (there is another function, `local-set-key`, that binds a key in a single buffer). Above, I have bound `C-c l` to the function `goto-line`. The key is described using a string. The special syntax "`\C-<char>`" means the Control key held down while the key `<char>` is pressed. Likewise, "`\M-<char>`" indicates the Meta key.

[9] Sometimes unofficially called "Elisp".

All very well, but how did I know that the function's name was "`goto-line`"? I may know that I want to bind `C-c l` to some function that prompts for a line number and then moves the cursor to that line, but how did I find out that function's name?

This is where Emacs's online help facilities come in. Once you have decided what kind of function you are looking for, you can use Emacs to track down its exact name. Here's one quick and dirty way to do it: since Emacs gives completion on function names, just type `C-h f` (which is `describe-function`, remember), and then hit Tab without typing anything. This asks Emacs to do completion on the empty string — in other words, the completion will match every single function! It may take a moment to build the completion list, since Emacs has so many internal functions, but it will display as much of it as fits on the screen when it's ready.

At that point, hit `C-g` to quit out of `describe-function`. There will be a buffer called "`*Completions*`" which contains the completion list you just generated. Switch to that buffer. Now you can use `C-s`, `isearch`, to search for likely functions. For example, it's a safe assumption that a function which prompts for a line number and then goes to that line will contain the string "`line`" in its name. Therefore, just start searching for the string "`line`", and you'll find what you're looking for eventually.

If you want another method, you can use `C-h a`, `command-apropos`, to show all functions whose names match the given string. The output of `command-apropos` is a little harder to sort through than just searching a completion list, in my opinion, but you may find that you feel differently. Try both methods and see what you think.

There is always the possibility that Emacs does not have any predefined function to do what you're looking for. In this situation, you have to write the function yourself. I'm not going to talk about how to do that — you should look at the Emacs Lisp library for examples of function definitions, and read the Info pages on Emacs Lisp. If you happen to know a local Emacs guru, ask her how to do it. Defining your own Emacs functions is not a big deal — to give you an idea, I have written 131 of them in the last year or so. It takes a little practice, but the learning curve is not steep at all.

Another thing people often do in their `.emacs` is set certain variables to preferred values. For example, put this in your `.emacs` and then start up a new Emacs:

```
(setq inhibit-startup-message t)
```

Emacs checks the value of the variable `inhibit-startup-message` to decide whether or not to display certain information about version and lack of warranty when it starts up. The Lisp expression above uses the command `setq` to set that variable to the value '`t`', which is a special Lisp value that means **true**. The opposite of '`t`' is '`nil`', which is the designated **false** value in Emacs Lisp. Here are two things that are in my `.emacs` that you might find useful:

```
(setq case-fold-search nil) ; gives case-insensitivity in searching
;; make C programs indent the way I like them to:
(setq c-indent-level 2)
```

The first expression causes searches (including `isearch`) to be case-insensitive; that is, the search will match upper- or lower-case versions of a character even though the search string contains only the lower-case version. The second expression sets the default indentation for C language statements to be a little smaller than it is normally — this is just a personal preference; I find that it makes C code more readable.

The comment character in Lisp is "`;`". Emacs ignores anything following one, unless it appears inside a literal string, like so:

```
;; these two lines are ignored by the Lisp interpreter, but the
;; s-expression following them will be evaluated in full:
(setq some-literal-string "An awkward pause; for no purpose.")
```

It's a good idea to comment your changes to Lisp files, because six months later you will have no memory of what you were thinking when you modified them. If the comment appears on a line by itself, precede it with two semicolons. This aids Emacs in indenting Lisp files correctly.

You can find out about internal Emacs variables the same ways you find out about functions. Use `C-h v`, `describe-variable` to make a completion list, or use `C-h C-a`, `apropos`. `Apropos` differs from `C-h a`, `command-apropos`, in that it shows functions and variables instead of just functions.

The default extension for Emacs Lisp files is "`.el`", as in "`c-mode.el`". However, to make Lisp code run faster, Emacs allows it to be **byte-compiled**, and these files of compiled Lisp code end in "`.elc`" instead of "`.el`". The exception to this is your `.emacs` file, which does not need the `.el` extension because Emacs knows to search for it on startup.

To load a file of Lisp code interactively, use the command `M-x load-file`. It will prompt you for the name of the file. To load Lisp files from inside other Lisp files, do this:

```
(load "c-mode") ; force Emacs to load the stuff in c-mode.el or .elc
```

Emacs will first add the `.elc` extension to the filename and try to find it somewhere in the `load-path`. If it fails, it tries it with the `.el` extension; failing that, it uses the literal string as passed to `load`. You can byte-compile a file with the command `M-x byte-compile-file`, but if you modify the file often, it's probably not worth it. You should never byte-compile your `.emacs`, though, nor even give it a `.el` extension.

After your `.emacs` has been loaded, Emacs searches for a file named `default.el` to load. Usually it's located in a directory in load-path called `site-lisp` or `local-elisp` or something (see the example `load-path` I gave a while ago). People who maintain Emacs on multi-user systems use default.el to make changes that will affect everyone's Emacs, since everybody's Emacs loads it after their personal `.emacs`. `Default.el` should not be byte-compiled either, since it tends to be modified fairly often.

If a person's `.emacs` contains any errors, Emacs will not attempt to load `default.el`, but instead will just stop, flashing a message saying "`Error in init file.`" or something. If you see this message, there's probably something wrong with your `.emacs`.

There is one more kind of expression that often goes in a `.emacs`. The Emacs Lisp library sometimes offers multiple packages for doing the same thing in different ways. This means that you have to specify which one you want to use (or you'll get the default package, which is not always the best one for all purposes). One area in which this happens is Emacs's Scheme interaction features. There are two different Scheme interfaces distributed with Emacs (in version 19 at least): `xscheme` and `cmuscheme`.

```
prompt> ls /usr/lib/emacs/19.19/lisp/*scheme*
/usr/lib/emacs/19.19/lisp/cmuscheme.el
/usr/lib/emacs/19.19/lisp/cmuscheme.elc
/usr/lib/emacs/19.19/lisp/scheme.el
/usr/lib/emacs/19.19/lisp/scheme.elc
/usr/lib/emacs/19.19/lisp/xscheme.el
/usr/lib/emacs/19.19/lisp/xscheme.elc
```

I happen to like the interface offered by `cmuscheme` much better than that offered by `xscheme`, but the one Emacs will use by default is `xscheme`. How can I cause Emacs to act in accordance with my preference? I put this in my `.emacs`:

```
;; notice how the expression can be broken across two lines.  Lisp
;; ignores whitespace, generally:
```



```
(autoload 'run-scheme "cmuscheme"
"Run an inferior Scheme, the way I like it." t)
```

The function `autoload` takes the name of a function (quoted with "`'`", for reasons having to do with how Lisp works) and tells Emacs that this function is defined in a certain file. The file is the second argument, a string (without the "`.el`" or "`.elc`" extension) indicating the name of the file to search for in the `load-path`.

The remaining arguments are optional, but necessary in this case: the third argument is a documentation string for the function, so that if you call `describe-function` on it, you get some useful information. The fourth argument tells Emacs that this autoloadable function can be called interactively (that is, by using `M-x`). This is very important in this case, because one should be able to type `M-x run-scheme` to start a scheme process running under Emacs.

Now that `run-scheme` has been defined as an autoloadable function, what happens when I type `M-x run-scheme`? Emacs looks at the function `run-scheme`, sees that it's set to be autoloaded, and loads the file named by the autoload (in this case, "`cmuscheme`"). The byte-compiled file `cmuscheme.elc` exists, so Emacs will load that. That file *must* define the function `run-scheme`, or there will be an autoload error. Luckily, it does define `run-scheme`, so everything goes smoothly, and I get my preferred Scheme interface[10].

An `autoload` is a like a promise to Emacs that, when the time comes, it can find the specified function in the file you tell it to look in. In return, you get some control over what gets loaded. Also, autoloads help cut down on Emacs's size in memory, by not loading certain features until they are asked for. Many commands are not really defined as functions when Emacs starts up. Rather, they are simply set to autoload from a certain file. If you never invoke the command, it never gets loaded. This space saving is actually vital to the functioning of Emacs: if it loaded every available file in the Lisp library, Emacs would take twenty minutes just to start up, and once it was done, it might occupy most of the available memory on your machine. Don't worry, you don't have to set all these autoloads in your `.emacs`; they were taken care of when Emacs was built.

4.14 Finding Out More

I have not told you everything there is to know about Emacs. In fact, I don't think I have even told you 1% of what there is to know about Emacs. While you know enough to get by, there are still lots of time-saving tricks and conveniences that you ought to find out about. The best way to do this is to wait until you find yourself needing something, and then look for a function that does it.

The importance of being comfortable with Emacs's online help facilities cannot be emphasized enough. For example, suppose you want to be able to insert the contents of some file into a buffer that is already working on a different file, so that the buffer contains both of them. Well, if you were to guess that there is a command called `insert-file`, you'd be right. To check your educated guess, type `C-h f`. At the prompt in the minibuffer, enter the name of a function that you want help on. Since you know that there is completion on function names, and you can guess that the command you are looking for begins with "insert", you type insert and hit Tab. This shows you all the function names that begin with "insert", and "insert-file" is one of them.

So you complete the function name and read about how it works, and then use `M-x insert-file`. If you're wondering whether it's also bound to a key, you type `C-h w insert-file` Return, and find out. The more you know about Emacs's help facilities, the more easily you can ask Emacs questions about itself. The ability to do so, combined with a spirit of exploration and a willingness to learn new ways of doing things, can end up saving you a lot of keystrokes.

[10] By the way, `cmuscheme` was the interface I was talking about earlier, in the section on working with Scheme, so if you want to use any of the stuff from that tutorial, you need to make sure that you run `cmuscheme`.

To order a copy of the Emacs user's manual and/or the Emacs Lisp Programming manual, write to:

Free Software Foundation
675 Mass Ave
Cambridge, MA 02139
USA

Both of these manuals are distributed electronically with Emacs, in a form readable by using the Info documentation reader (`C-h i`), but you may find it easier to deal with treeware than with the online versions. Also, their prices are quite reasonable, and the money goes to a good cause — quality free software! At some point, you should type `C-h C-c` to read the copyright conditions for Emacs. It's more interesting than you might think, and will help clarify the concept of free software. If you think the term "free software" just means that the program doesn't cost anything, please do read that copyright as soon as you have time!